L Number	Hits	Search Text	DB	Time stamp
1	72028	Process near5 type	USPAT	2003/09/03 09:33
2	72028	(Process near5 type)	USPAT	2003/09/03 09:33
3	0	("klapphotz-nicole.in.").PN.	USPAT; US-PGPUB	2003/09/03 10:29
4	0	("klappholz.in.").PN.	USPAT; US-PGPUB	2003/09/03 09:38
5	3445	Process Near3 class	USPAT; US-PGPUB	2003/09/03 09:39
6	114	(Process Near3 class) and parallel near5 process	USPAT; US-PGPUB	2003/09/03 10:29
8	11	(((Process Near3 class) and parallel near5 process) and lock$) and unlock$	USPAT; US-PGPUB	2003/09/03 09:41
7	30	((Process Near3 class) and parallel near5 process) and lock$	USPAT; US-PGPUB	2003/09/03 09:48
9	11	((Process Near3 class) and parallel near5 process) and unlock$	USPAT; US-PGPUB	2003/09/03 10:23
10	0	("L6andReadynear6queue").PN.	USPAT; US-PGPUB	2003/09/03 10:30
11	0	("L6andReadynear6queue").PN.	USPAT; US-PGPUB	2003/09/03 10:31
12	1454	((Process Near3 class) and parallel near5 process) ready near5 queue	USPAT; US-PGPUB	2003/09/03 10:31
13	664	((Process Near3 class) and parallel near5 process) timer near5 queue	USPAT; US-PGPUB	2003/09/03 10:32

L Number	Hits	Search Text	DB	Time stamp
1	72028	Process near5 type	USPAT	2003/09/03 12:44
2	72028	(Process near5 type)	USPAT	2003/09/03 09:33
3	0	("klapphotz-nicole.in.").PN.	USPAT; US-PGPUB	2003/09/03 10:29
4	0	("klappholz.in.").PN.	USPAT; US-PGPUB	2003/09/03 09:38
5	3445	Process Near3 class	USPAT; US-PGPUB	2003/09/03 09:39
6	114	(Process Near3 class) and parallel near5 process	USPAT; US-PGPUB	2003/09/03 10:29
8	11	(((Process Near3 class) and parallel near5 process) and lock$) and unlock$	USPAT; US-PGPUB	2003/09/03 09:41
7	30	((Process Near3 class) and parallel near5 process) and lock$	USPAT; US-PGPUB	2003/09/03 09:48
9	11	((Process Near3 class) and parallel near5 process) and unlock$	USPAT; US-PGPUB	2003/09/03 10:23
10	0	("L6andReadynear6queue").PN.	USPAT; US-PGPUB	2003/09/03 10:30
11	0	("L6andReadynear6queue").PN.	USPAT; US-PGPUB	2003/09/03 10:31
12	1454	((Process Near3 class) and parallel near5 process) ready near5 queue	USPAT; US-PGPUB	2003/09/03 10:31
13	664	((Process Near3 class) and parallel near5 process) timer near5 queue	USPAT; US-PGPUB	2003/09/03 10:32
14	11	(Process near5 type) same time adj slice$	USPAT	2003/09/03 12:16
15	11	(Process near5 type) same(queue and class)	USPAT	2003/09/03 12:53
16	726	(709/100).CCLS.	USPAT; US-PGPUB	2003/09/03 12:54
17	0	("L16andprocessnear5type").PN.	USPAT; US-PGPUB	2003/09/03 12:55
18	90045	((709/100).CCLS.) ans process near5 type	USPAT; US-PGPUB	2003/09/03 12:55
19	72	((709/100).CCLS.) and process near5 type	USPAT; US-PGPUB	2003/09/03 13:07
20	3	(((709/100).CCLS.) and process near5 type) and time adj slice	USPAT; US-PGPUB	2003/09/03 12:59
21	0	(((709/100).CCLS.) and process near5 type) and Execution adj time adj slice	USPAT; US-PGPUB	2003/09/03 12:59
22	20	((709/100).CCLS.) and process near type	USPAT; US-PGPUB	2003/09/03 13:11
23	12	((709/100).CCLS.) and ready adj queue	USPAT; US-PGPUB	2003/09/03 13:11

L Number	Hits	Search Text	DB	Time stamp
1	0	((Process Near3 class) and parallel near5 process) and lock adj procedure	USPAT; US-PGPUB	2003/09/03 14:38
2	0	Process near5 type same (lock adj procedure)	USPAT	2003/09/03 14:39
3	446	Process near5 type same lock$	USPAT	2003/09/03 14:39
4	137	(Process near5 type same lock$) and procedure	USPAT	2003/09/03 14:41
5	21	"L14" same (flag)	USPAT	2003/09/03 14:42
6	21	"L14" same (flag)	USPAT	2003/09/03 14:42

L Number	Hits	Search Text	DB	Time stamp
1	0	((Process Near3 class) and parallel near5 process) and lock adj procedure	USPAT; US-PGPUB	2003/09/03 14:38
2	0	Process near5 type same (lock adj procedure)	USPAT	2003/09/03 14:39
3	446	Process near5 type same lock$	USPAT	2003/09/03 14:39
4	137	(Process near5 type same lock$) and procedure	USPAT	2003/09/03 14:48
5	21	"L14" same (flag)	USPAT	2003/09/03 14:42
6	21	"L14" same (flag)	USPAT	2003/09/03 14:42
7	1313	709/100	USPAT	2003/09/03 14:52
8	329	709/100 and Lock$	USPAT	2003/09/03 14:52

US005465335A

United States Patent [19]

Anderson

[11] Patent Number: 5,465,335

[45] Date f Patent: Nov. 7, 1995

[54] **HARDWARE-CONFIGURED OPERATING SYSTEM KERNEL HAVING A PARALLEL-SEARCHABLE EVENT QUEUE FOR A MULTITASKING PROCESSOR**

[75] Inventor: **Mark F. Anderson**, Biose, Id.

[73] Assignee: **Hewlett-Packard Company**, Palo Alto, Calif.

[21] Appl. No.: **258,919**

[22] Filed: **Jun. 13, 1994**

Related U.S. Application Data

[63] Continuation of Ser. No. 776,931, Oct. 15, 1991, abandoned.

[51] **Int. Cl.**[6] **G06F 7/00**

[52] **U.S. Cl.** **395/375**; 395/650; 395/800; 364/DIG. 1:DIG. 2

[58] **Field of Search** 395/800, 650, 395/375

[56] **References Cited**

U.S. PATENT DOCUMENTS

4,084,228	4/1978	Dufond et al.	395/650
4,374,409	2/1983	Bienvenu et al.	395/650
4,387,427	6/1983	Cox et al.	395/650
4,658,351	4/1987	Teng	395/650
4,908,750	3/1990	Jablow	395/650
4,965,716	10/1990	Sweeny	395/650
5,012,409	4/1991	Fletcher et al.	395/650
5,185,871	2/1993	Frey et al.	395/375
5,237,684	8/1993	Record et al.	395/650
5,291,614	3/1994	Baker et al.	395/800

FOREIGN PATENT DOCUMENTS

63-208948 8/1988 Japan .

OTHER PUBLICATIONS

Structured Computer Organization by Andrew S. Tanenbaum, Prentice—Hall, Inc. 1984, pp. 10–12.

Test and Evaluation of the SVID–Compliant REAL/IX Real Time Operating System by Zuccarelli et al, IEEE 1990, pp. 81–85.

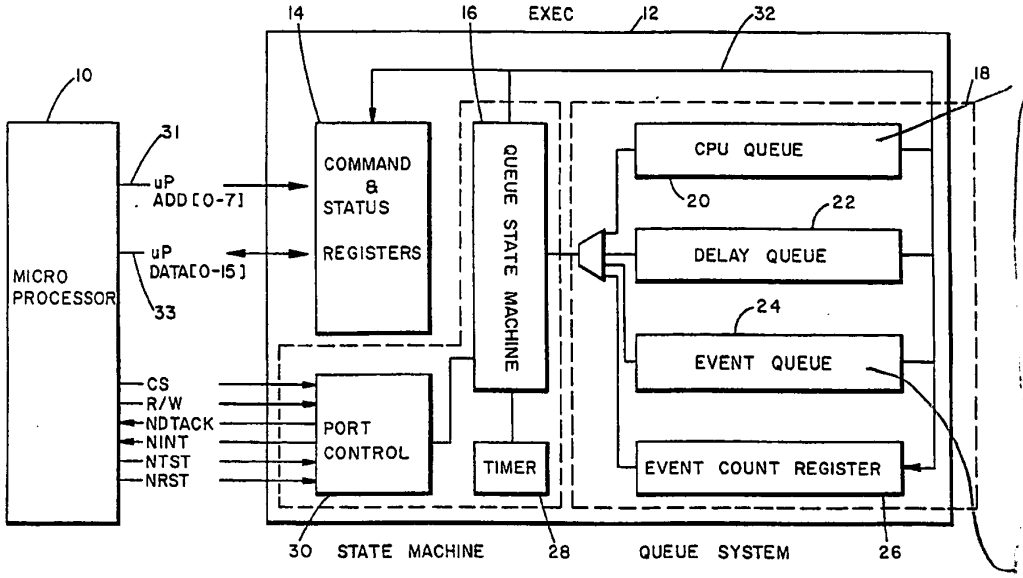
FASTCHART—A Fast Time Deterministic CPU and Hardware Based Real–Time–Kernel by Lennant Linch et all, IEEE Publication, Jun. 1991, pp. 36–40.

Primary Examiner—David Y. Eng
Assistant Examiner—Zarni Maung

[57] **ABSTRACT**

A multitasking data processing system is provided with a hardware-configured operating system kernel. The system includes a processor queue that includes a plurality of word stores, each word store storing a task name, in execution priority order, that is ready for processing. An event queue in the kernel includes a plurality of word stores for storing task names that await the occurrence of an event to be placed in the processor queue. When an associated processor signals the occurrence of an event, matching logic searches all word stores in the event queue, in parallel, to find a task associated with the signalled event and then transfers the task to the processor queue. Shift logic is also provided for simultaneously transferring a plurality of task names, in parallel, in the processor queue to make room for a task name transferred from the event queue.

17 Claims, 7 Drawing Sheets



Ready Queue
Word store
+
Task name
+
Priority order
Ready for processing

Timer queue

HARDWARE-CONFIGURED OPERATING SYSTEM KERNEL HAVING A PARALLEL-SEARCHABLE EVENT QUEUE FOR A MULTITASKING PROCESSOR

CROSS REFERENCE TO RELATED APPLICATION

This is a continuation of application Ser. No. 07/776,931 filed on Oct. 15, 1991, now abandoned.

FIELD OF THE INVENTION

This invention relates to processors that are programmed to operate on many tasks in parallel, and more particularly, to a hardware implementation of a portion of the processor's operating system to enable multitasking operations to proceed more rapidly.

BACKGROUND OF THE INVENTION

Real-time data processors are able to handle the processing of a number of tasks on a concurrent basis. Multiprocessors perform this function by providing a number of processors that operate on the tasks in parallel. Single processor systems, operating in real-time, handle "parallel" processing on a multitasking basis. Multitasking is a process of executing several tasks concurrently or in parallel. The concurrently processed tasks execute logically at the same time, even though they may not execute physically at the same time.

Multitasking operations perform under control of the computer's operating system and, in particular, under control of an "Executive" segment thereof that controls the scheduling of the concurrently running tasks. The Executive segment (hereinafter called EXEC) provides an interface between the tasks and the central processing unit. To each task, the EXEC appears as the task's own central processing unit. The EXEC handles all of the details involved in sharing the physical CPU with all of the tasks in the system. Thus, an EXEC controls which task should have possession of the CPU at any time by examining priority and readiness levels assigned to each task.

In general, the EXEC enables the running of one task on the CPU until another request to use the CPU is received from a higher priority task. The running task is "suspended", and the higher priority task is allowed to run. This process may occur many times over but sooner or later the highest priority task will complete its processing and voluntarily suspend itself. Thus, even though a lower priority task may be suspended many times before its completion, it eventually completes and produces the same result as if it had run uninterrupted.

Tasks have various execution states. An active task is one which has control of the CPU and is executing. Only one task can be active at any given time on a multitasking CPU. An inactive task is not executing and is not waiting to execute. It is simply idle. A ready task is one which is waiting for CPU time to become available so that it can execute. A waiting task is one that has suspended operation until the occurrence of some event. The event can be generated by another task, or a hardware event. Upon occurrence of the event, the task is moved from the waiting state to either the ready or the active state depending upon the priority of currently active tasks.

Tasks may be synchronized by priority or by task readiness or a combination of both. In general, an EXEC includes a number of utility routines that are used by tasks to perform necessary control functions. Under these utility routines, any task may schedule another task, suspend itself or another task, signal an event, wait for an event or delay an event for a period of time. The principle EXEC utilities are as follows: Schedule, Suspend, Signal, Wait and Delay. The Schedule utility is used by an active task when it wants another task to begin or resume execution. The Schedule utility allows the highest priority ready task to execute. The Suspend utility is used by an active task to remove itself from the ready state or move another task to the inactive state. Control is then given to the current highest-priority ready task.

The Signal utility is used by the active task or an interrupt service routine to generate a particular event. If no task is waiting on the event that is signalled, the EXEC returns control to the active task. If another task is waiting on the signalled event, control returns to the highest priority task.

The Wait utility is used when the active task wishes to suspend execution and resume on an occurrence of a particular event, or events.

The Delay utility is used by the active task when it needs either itself or another task to wait a specific amount of time before resuming execution. This utility allows an asynchronous task to be synchronized using time as a reference.

An operating system incorporates an interrupt facility which is the primary means for synchronizing firmware operations with hardware events. When an interrupt occurs, control is transferred from whatever task is executing to an interrupt handling module. One interrupt module exists for each type of interrupt that can occur.

A variety of data structures are employed by the EXEC to implement the utilities above described. A CPU queue is a list of tasks in the ready state. The highest priority task is the currently active task. If there are two ready tasks of the same priority, the task that is ready first becomes the active task. An Event queue is a list of tasks in a waiting state. A Delay queue is another list of delayed events which, after the duration of a delay, will cause tasks waiting on these events to be moved to the CPU queue to become ready tasks.

The described queues are dynamic in structure and must accommodate at least four basic operations: insertion of an item; deletion of an item; location of an item; and modification of an item. Each queue may be configured as a linear list, a singly linked list, or a doubly linked list. A singly linked list is one where each item contains a pointer to the next item in the list. A doubly linked list has items with two linked fields. Each item in the list has a pointer to the next item to the right and to the next item left of it.

As above indicated, an EXEC is generally configured in software as a portion ("kernel") of the operating system. Often, the EXEC is required to perform queue-wide operations that include comparisons and shifting of the contents of the queue. Such operations are performed serially and their time of execution varies at least linearly with the length of the queue. Such execution times can have a deleterious affect on the performance of the operating system and presents real impediments to the improvement of the system's performance.

Accordingly, it is an object of this invention to provide a multitasking data processing system with an EXEC that is not constrained by serial processing operations.

It is another object of this invention to provide an EXEC that enables more efficient operation of a multitasking data

processing system.

It is another object of this invention to provide a multitasking data processing system with a hardware implemented operating system kernel wherein queues maintained in the kernel require no separate priority indication.

SUMMARY OF THE INVENTION

A multitasking data processing system is provided with a hardware-configured operating system kernel. The system includes a processor queue that includes a plurality of word stores, each word store storing a task name, in execution priority order, that is ready for processing. An event queue in the kernel includes a plurality of word stores for storing task names that await the occurrence of an event to be placed in the processor queue. When an associated processor signals the occurrence of an event, matching logic searches all word stores in the event queue, in parallel, to find a task associated with the signalled event and then transfers the task to the processor queue. Shift logic is also provided for simultaneously transferring a plurality of task names, in parallel, in the processor queue to make room for a task name transferred from the event queue.

DESCRIPTION OF THE DRAWINGS

FIG. **1** is a high level block diagram of a system incorporating the invention hereof.

FIG. **2** is a block diagram of a CPU queue used in the system of FIG. **1**.

FIG. **3** is a block diagram of each queue element within the CPU queue of FIG. **2**.

FIG. **4** is more detailed logic of a word store within a queue element word of FIG. **3**.

FIG. **5** is a circuit diagram of a content addressable memory cell.

FIG. **6** is as block diagram of a set of event count registers used with the invention.

FIG. **7** is a block diagram of a delay timer used with the invention.

FIG. **8** is an exemplary set of read/write registers contained within the command and status registers shown in FIG. **1**.

DETAILED DESCRIPTION OF THE INVENTION

In brief, this invention removes certain critical operating system functions from operating system software and implements them in a hardware structure. Through this mechanism, certain often-found data reorganization operations may be performed in parallel and in a predetermined time interval. The hardware kernel employs, in the main, five commands, i.e., Schedule, Suspend, Signal, Wait and Delay. Through these commands, and combinations thereof, a set of real-time operating system primitives are created which enable substantial improvement in an attached processor's operating parameters. Due to the hardware configuration of the queues in the kernel, queue-wide operations are easily accomplished in a fraction of the time needed for similar operations in a software environment.

EXEC MODULE (KERNEL)

Turning now to FIG. **1**, a high level block diagram is shown of the system. A microprocessor **10** has incorporated therewith, an EXEC module **12** which is implemented in hardware. EXEC module **12** contains three main functional blocks, i.e., a command and status register set **14**, a queue state machine **16** and a queue system **18**. Queue system **18** contains the data structures that all utilities manipulate. There are three physically identical queues in queue system **18**, i.e., CPU queue **20**, Delay queue **22** and Event queue **24**.

CPU queue **20** is a priority-ordered list of names of Task Control Blocks (TCBs) and associated events that cause the TCBs to be placed in the CPU queue. CPU queue **20** controls the sequence that the various tasks execute within microprocessor **10**. TCBs contain all the information pertaining to the status of a task and are used within microprocessor **10** to control such tasks. The name of each TCB is a value (e.g., from 0 to 255) which indicates its priority as well as designating the TCB. The priority of the task, referenced in the task's TCB name, is used to determine when a task is given possession of microprocessor **10**. The TCB at the "head" of CPU Queue **20** retains possession of microprocessor **10** for that task. If a task of higher priority is placed in the queue, the currently running task is replaced by the higher priority task at the head of CPU Queue **20** and the task of higher priority executes. At any time, the number of tasks' TCBs in CPU Queue **20** may range from all (all tasks ready to execute), to none (no active or ready tasks).

Event queue **24** contains a priority-ordered list of names of TCBs, with each TCB in the list joined to an event name, upon whose occurrence, the TCB will be moved to CPU Queue **20**. Delay Queue **22** contains a list of event names, each event name associated with a delay which indicates the amount of time before the associated event will be signalled. The event names in Delay Queue **22** are prioritized, based upon their associated delay values.

Queue system **18** also includes a set of event count registers **26** that keep track of the availability of a predetermined number of possible events that can occur within the system. It is to be recalled that an event can be signaled and waited on by system tasks. The number of events that can be accommodated by the system is dictated by the size of an address and is not to be considered a limitation of the system. For purposes of description it will be assumed that the system can accommodate up to 256 possible events and utilizes an 8-bit address for any such event.

Within command and status registers **14** is a command register that is written to by microprocessor **10** to start an EXEC utility. In addition, there are a number of registers that are "read-accessible" by microprocessor **10** and contain data produced by operations within EXEC module **12**. A list of those registers will be found below.

Queue state machine **16** controls the actions of EXEC module **12**. Upon receipt from microprocessor **10** of an EXEC command into a register in command and status registers **14**, queue state machine **16** executes the required actions on a queue or queues in queue system **18**. Queue state machine **16** also performs required actions on registers in command and status registers **14** or, further, controls a timer **28** and a port control module **30**. Port control module **30** provides I/O command functions for EXEC module **12**. Address and data connections between microprocessor **10** and EXEC module **12** are provided directly through command and status registers module **14** via lines **31** and **33**, respectively.

SIGNAL LINE DEFINITIONS.

Signal lines existing between microprocessor **10** and EXEC module **12** are as follows:

Inputs to EXEC Module 12

CS - chip select
R/W - read/not (write) line
uPAddr (0–7) - address lines
SCLK - EXEC system clock (not shown)
NRST - External Reset line
NTST - Test line (for test mode operation)

Bidirectional lines to EXEC Module 12

uP Data (0–15) - data lines

Outputs from EXEC Module 12

NDTACK - data transfer acknowledge (asynchronous acknowledge)
NINT - interrupt line to CPU

The following signals go between Queue State Machine **16** and each of queues **20, 22** AND **24** via bus **32** (but are not specifically illustrated in FIG. **1**).

D (**0–7**)—EXEC Internal Data Bus

ADDR (**0–15**)—EXEC Internal Address Bus

CMD (**0–3**)—Command Lines to Queue system **18**

QEN (**0–3**)—Queue Command Enables (one for each queue and array)

There are 256 queue elements in each queue. Each queue element stores two, 8-bit words. Therefore, **8** data lines and **9** address lines are required to access any word in any queue. The ninth address line is the CMD(**3**) signal line. CMD(**0–2**) are used to specify which of 8 different commands will be performed. CMD(**3**) determines which of two queue element words the operation will be performed on. QEN(**0–3**) lines are used to enable only one of the queues.

Commands can be sent to a particular queue to execute arithmetic comparisons. These arithmetic comparisons are done on all queue elements in parallel. The result of such comparisons is used to make parallel shifts in the queue in order to insert or delete items. This yields a significant speed improvement over manipulating these data as is typically done with pointers into a list. The following additional signals run from Queue State Machine **16** to the Event Count Registers **26** over Bus **32** (not specifically shown in FIG. **1**)

ECOP (**0–1**)—Event Counter Operations

ECOC—Event Counter Output Control

ECCLK—Event Counter Clock

ECLR—Event Register Array Clear

EOC—Event Register Output Control

These signals are used to control 256 8-bit registers in Event count registers **26** and a counter therein which can be loaded with any of these values and incremented or decremented.

Each register location in Event Count registers **26** is associated with a correspondingly numbered event. When a "Signal of a particular event occurs, the appropriate register value is read, incremented, and re-written. Likewise, when a Wait is executed on a particular event, the appropriate register's value is read, decremented, and re-written.

QUEUE STRUCTURES

Turning now to FIG. **2**, the basic structure of CPU queue **20** is illustrated. It will be recalled, that the hardware structure of each queue is identical. TCB names are stored in CPU queue **20** in priority order. Rather than a assigning a separate priority value to a TCB name, it has been determined that substantial storage area can be conserved by assigning as a TCB name, the actual priority value assigned to the TCB. Thus, a TCB having the name **0** has the highest priority value and is referred to by an address indication in EXEC module **12** by an all-0s address. Other TCBs of lower priority are similarly denoted.

CPU queue **20** contains a plurality of queue elements **35**. For illustration purposes, CPU queue **20** is shown with only four queue elements, instead of the 256 which can be used with an eight bit address. The structure of a queue element is shown in FIG. **3** and will be described in detail below. Data lines D(**0–7**) carry data into and out of data ports in each of queue elements **35** (see FIG. **2**). CMD(**0–3**) are address lines that are used to specify which of eight different commands will be performed by each queue element **35**. The CMD(**3**) line provides a level which determines which of two words in a queue element **35** the operation will be performed upon. QEN(**0**) is an enable line and enables one of the three queues (**20, 22, 24**).

An address bus ADDR (**0–7**) is decoded by address decoder **36** and is used to select an individual queue element **35** (via an output on cable **38** which contains 256 separate word lines). Each of the word lines in word bus **38** is connected to an individual queue element **35** so as to enable logical operations in any one chosen queue element in accordance with a received address into address decoder **36**. Such connections are not shown in FIG. **2** to avoid over-complication of the diagram.

Address bus ADDR (**0–7**) also carries address information back to queue state machine **16** during comparison operations. During such an operation, the address of a queue element **35** that meets a comparison criteria is returned on address bus ADDR (**0–7**).

A resolver/encoder **40** resolves the case where more than one queue element **35** meets a comparison criteria. For instance, if a command is issued to return an address of a queue element that contains "numeral 0" and all queue elements contain "0", then resolver encoder **40** causes address bus ADDR **0–7** to return the lowest numbered queue location to queue state machine **16**. In the example given, resolver encoder **40** would return location **1** on address bus ADDR (**0–7**).

Each queue element **35** contains two words of information pertaining to a task. In general, queue element **35** at position **1** in CPU queue **20** will contain two words pertaining to the highest-priority task awaiting action. Those words are the TCB name having the lowest numerical value (priority) and the name of the event that caused the TCB name to be moved into CPU queue **20**. Queue elements at position **3, 4** etc. will contain TCB names with lower priority (and higher numerical value).

There are two local buses **42** and **44** that connect each queue element **35** to its next higher-numbered neighbor queue element and are used when inserting or deleting items from a queue. Each queue element **35** has two comparison operations incorporated into its structure, the results of which are manifest at GT output **46** and M output **48**. The signals "GT" and "M" stand for "greater than" and "match", respectively. In brief, when data is applied to data bus D(**0–7**), each queue element **35** determines whether a data word contained within it matches in value or is greater in value in relation to the applied data word. If it finds that its stored word value is greater than the data bus value, then an output is impressed on output GT line **46**. If a match is found between the values, M output **48** is energized. A multiplexer **50** is connected to all queue elements **35** and is controlled to select either all "greater than" outputs **46** from queue elements **35** or all "match" outputs **48** from queue elements **35**. In the example shown in FIG. **2**, multiplexer **50** provides four outputs from the selected greater than or match outputs

from the four queue elements **35** shown. Which group of signals is selected is dependent on whether it is desired to know the lowest-numbered queue element **35** that matches the input data or the lowest-numbered queue element **35** that holds a value greater than the input data.

QUEUE ELEMENT STRUCTURE

Turning now to FIG. **3**, the structure of a queue element **35** is illustrated. In each queue element **35**, a pair of word stores **52** and **54** hold two 8-bit values, i.e., word A and word B. In addition, each word store contains the necessary logic to enable a comparison to be made between an incoming data word and the word contained therein. Details of that logic structure are shown in FIG. **4** and will be further considered below.

A queue element **35** holds the following information:

Queue Name	Word A	Word B
CPU Queue	Task Name/Priority	Event Case
Event Queue	Task Name/Priority	Event Name
Delay Queue	Delay Value	Event Name

The Task name/priority is manifest by a TCB name whose value is directly related to its priority (as above described). An "Event Case" is the event name value that occurred that caused the task denoted by a TCB in word A, to be moved into CPU queue **20**. An "Event Name" is a name or value given to a specific action within microprocessor **10**. For instance, an event name may be a value assigned to a hardware interrupt, an I/O interrupt, etc. A "Delay Value" is a value assigned to a time before an event is to occur.

Each word store **52, 54** has a GT and M output that is applied, respectively, to multiplexers **56** and **58**. The CMD (**3**) line is applied to multiplexers **56** and **58** and its level selects which word (word A or word B) is compared to provide the desired output on either GT line **46** or M line **48**.

LAST busses **42** and **44** and NEXT busses **42'** and **44'** are used to shift words between queue elements **35**. Each word store **52, 54** has its own LAST and NEXT bus that allows the contents of an entire queue element **35** to be shifted in one parallel operation. The contents of a word store **52** and a word store **54** always occupy the same queue element **35** together.

WORD STORE STRUCTURE

In FIG. **4**, the details of a word store **52** is illustrated. A multiplexer **70** receives four sources of input data for word store **52** (word A). Those sources are as follows:

- LAST A(**0–7**)—Contents of next lower Queue Element Word
- D(**0–7**)—EXEC Internal Data Bus
- NEXT A(**0–7**)—Contents of next higher Queue element Word A
- VDD—used to precharge BIT and nBIT (complement) lines during a comparison in a content-addressable memory **72**.

Content addressable memory (CAM) **72** contains eight, parallel connected bit cells for word A. Each bit cell provides both true and complement outputs for its respective bit. A pair of multiplexers **74** and **76** enable a comparison operation to be carried out with respect to the 8-bit word held in CAM **72**.

The details of a one-bit CAM cell are shown in FIG. **5**. Bit lines **82** and **84** are separate, but the word lines **81** and **83** of all eight one-bit CAM cells are tied together (not shown) to form a common 8-bit word line. The basic memory cell is a CMOS flip-flop **85** of known structure. To write into cell **85**, data is placed on bit line **82** and complement data is placed on complement bit line **84**. Then, word lines **81, 83** are asserted causing cell **80** to switch in accordance with the applied inputs. A read operation commences by precharging bit and bit complement lines **82** and **84**. Word lines **81, 83** are then asserted and bit line **82** manifests the value of the cell.

For a comparison operation, true data against which comparison is to be made is applied to complementary bit line **84** and complement data is placed on bit line **82**. If the applied data matches that in cell **85**, then a match transistor **86** remains in a nonconductive state, thereby enabling match line **88** to be unaffected by the state in cell **80**. There is one match line **88** in each memory cell.

Returning to FIG. **4**, a queue element logic module **73** receives a plurality of commands over CMD(**0–2**) lines decodes each command using wired logic and accordingly energizes one or more of its output lines **S1–S5**, E and W. The output lines from queue element logic module **73** are applied to multiplexers **70, 74, 75**, content addressable word memory **72** and a de-multiplexer **76** to enable a received command to be performed.

A comparator **77** examines the Match line outputs from CAM **72** and data inputs appearing on data bus D(**0–7**) to determine if received data has a greater value than that stored in word store **72**. Comparator **77** includes an exclusive NOR input stage (not shown) which reconstructs the stored data word from the Match line outputs and enables a subsequent magnitude comparison of the input values. If the input data is less in value than the stored word, the GT output from comparator **77** is asserted. If the input data is equal in value to the stored word, AND gate **78** asserts its M output.

QUEUE COMMANDS

Each word store **52/54** responds to the following commands:

- Match=—return highest-numbered location containing the compared word
- Match>—return highest-numbered location containing the next higher value than the compared word
- SR—shift contents of queue right (to next higher-numbered location)
- SL—shift contents of queue left (to next lower-numbered location)
- RD—read contents of addressed Word Store onto D(**0–7**)
- LD—load D(**0–7**) into addressed Word Store
- Init—initialize all queue element words to value=all 1's.

The "Match" commands are comparison commands and are executed in every queue element in a queue with respect to a selected word store (word A or word B). The RD and LD commands are only executed on one addressed word in a queue. The SR and SL commands both contain addresses which indicate a queue element word store from which data shifting is to occur. Thus, SR and SL shift commands only execute within queue elements whose address location is equal to or greater than the specified address. The selective shift operation is enabled by feeding the specified address from Queue State machine **16** to address decoder **36** (FIG. **2**) in the respective queue. Address decoder **36** responds to the address by activating the required word lines on bus **38** which feed into each queue element (e.g., Word (**0**) which feeds into queue element logic module **73** in FIG. **4**).

The Init command is used to initialize all word stores in a queue to the **1** state. This value indicates that a queue element is empty.

Returning to FIG. **4**, the operations of Word store **52** in

response to the Match =, RD, LD and SR commands will be considered.

The Match =command is initiated by appropriate values being emplaced on CMD(0–2) and QEN(0) lines flowing into queue element logic module 77. Those values indicate a Match =command for the specific queue in which word store 52 is present. The data value to be matched is presented to word store 52 on the D(0–7) lines from the data bus. Queue element logic module 73 activates output signals S1 and S2 to cause multiplexers 70 to pass the D(0–7) inputs to multiplexers 74, 75. Signal S3 from queue element logic module 73 is asserted and selects D(0–7) to flow to NBIT(0–7) input to word store 72. Signal S3 further causes the complement of D(0–7) to flow to the BIT(0–7) input to word store 72. If D(0–7) matches the value held in word store 72, resulting outputs on the Match lines cause AND gate 78 to assert Match output M.

Assume now that a Read Contents (RD) instruction appears over the CMD(0–2) lines accompanied by a QEN(0) enable signal. Queue element logic module 73 interprets these levels as an RD-A (read) word A command. An applied level to word(O) input to queue element logic module 73 indicates that this particular queue element's word A will be read onto data bus D(0–7). The read operation is initiated by pre-charging (raising to the logical 1 state) the bit lines to each bit cell in word store 72. This is accomplished by tri-stating outputs from multiplexers 74 and 76 with an assertion of the E line from queue element logic module 73. The word line to each bit cell in word store 72 is then asserted and the 8-bit values stored therein are read onto data bus D(0–7) through demultiplexer 76 under control of signals S4 and S5 from queue element logic module 73.

A Load (LD) command is commenced by command values applied onto CMD(0–2) inputs to queue element logic module 73, accompanied by a QEN(0) enable signal. An assertion of the word(O) input to queue element logic module 73 indicates that this particular queue element word A will be loaded from data bus D(0–7). In response to the LD command, queue element logic module 73 asserts signals S1 and S2 to multiplexer 70 which, in turn, causes the D(0–7) inputs to be connected to the input of multiplexers 74 and 75. Levels asserted on the S3 and S4 outputs from queue element logic module 73 then select the data values D(0–7) and their complements to be connected to bit (0–7) and NBIT (0–7) inputs to word store 72, respectively. Word line W from queue element logic module 73 is then asserted and causes storage of the data values into word store 72.

To accomplish a Shift Right command (SR), the CMD(0–2) lines are asserted with the proper command values accompanied by an enable signal on QEN(0). Word line word (0) is asserted indicating that this particular queue element word A will be part of a partial or total queue position shift. An SR (or SL) command is a concatenation of the above described RD and LD commands. The only differences are logic signals S4 and S5 from queue element logic module 73 select NEXTA(0–7) as outputs of the read function and the S1 and S2 outputs select LASTA(0–7) as inputs for the load (for the SR command).

A parallel operation Shift Right is achieved by reading all queue element words simultaneously onto the separate NEXTA buses, and then simultaneously loading all queue element words from their separate LASTA buses. While not expressly shown, each queue elements NEXTA bus is connected to the next higher-position queue element's LASTA bus. From the above, it can thus be seen that shifts of data between queue elements occur in parallel and simultaneously and may occur between selected queue elements or between all queue elements in a queue.

EVENT COUNT REGISTERS

EXEC module 12 includes 256 event count registers 26, one for each possible event (as limited by an 8-bit address). In FIG. 6, an 8-bit event counter 100 and the first three event count registers 102, 104 and 106 are shown. Event counter 100 is programmable and responds to either an event count being loaded via data bus D(0–7) or to an event count from one of registers 102, 104 or 106.

Each event count register has an assigned value indicative of one of 256 events which can occur in the system. If a register indicates a plus count, that is an indication that multiple tasks are waiting for the event to occur. If the event count register indicates a negative value, the indication is that the event has been signalled (occurred) more times than there are tasks waiting for the event's occurrence. Upon the occurrence of an event, the event count register corresponding to that event is examined to see the state of its count. If the count is seen to be positive, then queue state machine 16 knows that a task is present in event queue 24 and is awaiting the occurrence of the specific event. In such a case, event queue 24 is searched, in parallel, to find all TCBs that specify the specific event. The highest priority TCB that specifies the event is then chosen for execution. Queue state machine 16 transfers the chosen TCB from event queue 24 to CPU queue 20, where it is placed in priority order. When the task is removed from event queue 24, the value in the event count register is decremented through the action of counter 100.

When an event occurs, the Signal utility causes the value of the corresponding event count register to be read and used to program counter 100. Then, counter 100 decrements the value, which decremented value is then written back into an appropriate register. A Wait utility causes the same sequence to occur with the exception that the value is decremented. The value of any event count register (102,104, 106, etc.) is available to be read by queue state machines 16 via data bus D(0–7). The following command lines from queue state machine 16 are applied to counter 100.

ECOP(0–1) are used to specify whether counter 100 should increment, decrement, load, or clear.

ECOC is used to enable the output from counter 100 onto the data lines D(0–7)

ECCLK is the clock used to cause an operation on counter 100.

ECLR is used to initialize all event count registers 102, 104, 106, etc. to 0.

DELAY TIMER

In FIG. 7, the details of delay timer 28 in FIG. 1 are shown. Delay timer 28 counts system clock cycles and causes queue state machine 16 to signal any event whose delay is up. The number of clock cycles in a delay unit (i.e. a unit of delay time) is programmable and is held in a timer interval register within command and status registers 14. The delay unit value comes into an 8-bit counter 110 in delay timer 28 on TIR(0–7) lines from the timer interval register. 8-bit counter 112 counts the number of delay units by accumulating the number of clock cycles in each delay unit and then incrementing to a next delay unit count. 8-bit counter 112 counts up to 255 and then rolls over to 0. The Delay utility delays events by a relative time, not an absolute time.

COMMAND AND STATUS REGISTERS

With reference to FIG. 1, command and status registers comprise a number of microprocessor-accessible registers which receive 16 bit data (and commands) from microprocessor 10 and 8-bit data from various of the elements within

EXEC module **12**. All registers are 16 bits in width and each register is either readable or writeable, but not both. For instance, in FIG. **8**, examples of a write register **120** and read register **122** are shown along with their interconnections and control circuitry. Register **122** is a read register and receives data from EXEC module data bus D(**0–7**). Such data may subsequently be read out to microprocessor **10** upon application of enable signal ROEN(**5**). Similarly, write register **120** receives data from microprocessor **10** and upon application of an enable signal RIEN(**0**), provides such data as an output on data bus D(**0–7**).

The following is a list of microprocessor accessible registers (not shown) within command and status registers **14** and a description of the respective register's operation during execution of an associated utility operation.

Schedule Register (a Write Register): When a Task TCB is written to this register, the TCB is placed in CPU Queue **20** according to its priority (TCB value).

Suspend Register (a Write Register): When a Task TCB is written to this register, the TCB is removed from CPU Queue **20**; if it is indeed in CPU Queue **20**. If the TCB is not in CPU Queue **20**, an interrupt condition is generated to microprocessor **10**. Status indicating this condition is set in a Status Register.

Signal Register (a Write Register): When an Event control block is written to this register from microprocessor **10**, this event is signalled. If no task is waiting for this event, the only action taken is to decrement the Event Count for this event. If tasks are waiting for this event, then the TCB with the highest priority (lowest TCB value) is removed from Event Queue **24** and placed in CPU Queue **20**.

Wait Register (a Write Register): When an Event control block is written to this register from microprocessor **10**, the running task's TCB is placed in Event Queue **24** to wait for the named event. The running task's TCB is found at the first queue position in CPU Queue **20** (position **0**). This TCB is removed from CPU Queue **20** and placed in Event Queue **24** according to its priority (TCB value).

Delay Register (a Write Register): When an Event control block and an 8-bit delay value is written to this register from microprocessor **10**, the Event control block name is placed into Delay Queue **22**, prioritized by its delay value. The lower the delay, the closer to the head of the queue this block name is placed.

Status Register (a Read Register): The Status register contains information on any abnormal conditions that occur within the EXEC chip.

Timer Interval Register (a Write Register): The user writes a 16-bit value to this register which represents the number of system clocks that make up each delay value.

CPUQ Register, Event Q Register, Delay Q Register (Read Registers): These registers, when read by microprocessor **10**, will sequentially give the contents of all the Queue Element Words in the appropriate queue. These are Diagnostic Registers.

Diagnostic Pointer Register (a Write Register): When written to with a value, the diagnostic registers above when read will be set to read from this location within the queues.

Active TCB Register (a Read Register): When read, the TCBName of the current task that has highest priority in the CPU Queue is returned.

Event Case Register (a Read Register): When read, the Event control block Name of the event that caused a task to resume execution is returned. This value only has meaning if this task has executed a Wait utility call.

QUEUE STATE MACHINE

Queue state machine **16** controls the workings of EXEC Module **12**. Queue state machine **16** is, in turn, controlled by internal hardware that responds to commands written to command and status registers **14**, to sequentially energize control lines to cause system operations in accordance with a specified command. In all respects queue state machine **16** is conventional and its arrangement is known to those skilled in the art.

EXEC MODULE OPERATION-SCHEDULE UTILITY

Assume that microprocessor **10** writes a TCB of a task that it wishes to schedule, to the Schedule Register within command and status registers **14** in EXEC module **12**. The TCB value is thereby latched within the Schedule register. In response, queue state machine **16** performs a parallel search of CPU queue **20** to find a stored next lower priority TCB from the TCB being scheduled. Once found, the TCBs within CPU **20** are shifted one position to the right, starting from the incoming TCB's queue position. The position within CPU **20** vacated by this rightward shift is loaded with the TCB of the task to be scheduled.

EXEC module **12** now asserts a handshake signal NDTACK from Port control **30** to indicate that the transaction may now be terminated. In response, microprocessor **10** de-asserts its CS signal to Port control **30**, which signal precedes every transaction with EXEC module **12**.

CPU queue **20** has now been modified so that the TCB name of the task to be scheduled is present in CPU queue **20** in its proper priority position. Microprocessor **10**, to determine what is now the highest-priority task awaiting execution, reads the Active TCB register from command and status registers **14**. Microprocessor **10** may also read the Event case register. The access by microprocessor **10** to the Active TCB register causes queue state machine **16** to read the TCB name from the first queue element (highest priority) in CPU queue **20** and to place this value onto microprocessor data lines DATA (**0–15**). Port control **30** then asserts the NDTACK line to indicate that the value of the active TCB register is available to microprocessor **10**.

It will be recalled that the Event case is the name of the event that caused a name of a task to be moved into CPU queue **20**. Microprocessor **10** can access the Event case register to determine the event value within CPU queue **20**. This access causes the event name to be read from the first queue element in CPU queue **20** into the Event case register and thence to be placed on microprocessor **10**'s data bus DATA(**0–15**). The signal NDTACK is asserted indicating to microprocessor **10** that the value of this register is available on the output data bus.

QUEUE STATE MACHINE OPERATION—SCHEDULE UTILITY

The following are states that occur during a Schedule utility. Inputs to Queue state machine **16** are in lower case and outputs are uppercase. All events are synchronized to the system clock. For each state that is not exited until a particular event occurs, the event is indicated. If no event is specified for a given state, the state is exited upon the next state machine clock.

1. IDLE—EXEC chip select (cs) is not asserted, ntst is not asserted, nrst is not asserted, timer carry out (tco) is not asserted. Event—Chip select (cs) is asserted, CPU wishes to make an operating system call.
2. CMD RCVD—an EXEC command is received. Actions—The schedule Register latches the value the CPU provides on the uP Address lines. This value is the TCBName of the task that is to be scheduled for execution.
3. SETUP MATCH CMD—Begin the Schedule utility. Setup for a "Match>." command. Actions—CMD(**0–3**) lines set up with the command "Match>A." D(**0–7**)

lines get the value in the Schedule Register. QEN(0–3) is set to the CPUQ value to indicate that this action takes place on the CPU Queue.

4. FINISH MATCH CMD—Assert the "MATCH" line. Actions—De-assert last state's actions and assert the MATCH line.

5. READ MATCH>ADD—Read address lines for highest priority match>address. Actions—Store the value on ADDR(0–7) in a temporary register, and de-assert the MATCH line.

6. SETUP SR CMD—Setup for a shift right command. Actions—Put value in temporary register out on ADDR(0–7) lines. Set CMD(0–3) lines to "SR" (Shift Right) command, and set QEN(0–3) lines equal to CPUQ value (to indicate action on CPU Queue) .

7. FINISH SR CMD—Finish the Shift Right command. Actions—De-assert last actions.

8. SETUP LD CMD—Setup for a "LD" (load) command. Actions—CMD(0–3) lines gets the "LDA" command value. D(0–7) lines gets the contents of the Schedule Register. ADDR(0–7) lines get the value in the temporary register. QEN(0–3) lines are set equal to the CPUQ value (to indicate action on the CPU Queue).

8. FINISH LD CMD—Finish the load command. Actions—De-assert the actions of the last state.

10. SETUP END uP XACTION (Setup to end transaction with CPU. Actions—DTACK (data transfer acknowledge) is asserted. Event—cs is de-asserted.

11. END uP XACTION—End transaction with CPU. Actions—DTACK (data transfer acknowledge) is de-asserted.

12. IDLE—GO to state 1 above.

At this time, the task's TCB has been placed in the CPU Queue. In order to use the updated queue information, microprocessor **10** must execute a "$READ_{13}$ACTIVETCB" command. In addition, microprocessor **10** may execute a "$READ_{13}$EVENTCASE" in order to determine the event that caused this task to resume execution. These actions are performed for any utility call including Schedule, Suspend, Signal, Wait, and Delay.

$READ_{13}$ACTIVETCB Command

1. IDLE—EXEC chip select (cs), ntst, nrst, and tco (timer carry out) all are de-asserted.

2. SETUP RD CMD—Setup for a "RD" (read) command. Actions—CMD(0–3) lines get the command "RDA" (read word A). The QEN(0–3) lines get the CPUQ value. ADDR(0–7) lines are set equal to 0 (for the first location in the CPU Queue—the Queue Head).

3. WRITE ACTIVETCB REG—Write the Active TCB Register with the TCBName at the head of the CPU Queue. Actions—The Active TCB Register is written with the value now appearing on the D(0–7) lines. This is the TCBName of the next task to have use of the CPU resource by virtue of its priority.

4. SETUP END uP XACTION—Setup to end transaction with CPU. Actions—DTACK (data transfer acknowledge) is asserted. Event—cs is de-asserted.

5. END uP XACTION—End transaction with CPU. Actions—DTACK (data transfer acknowledge) is de-asserted.

6. IDLE (state 1 above)

READ_EVENTCASE Command

1. IDLE—EXEC chip select (cs), ntst, nrst, and tco (timer carry out) all are de-asserted.

2. SETUP RD CMD—Setup a "RD" (read) command. Actions—CMD(0–3) is set to the "RDB" (read word B) value. QEN(0–3) is set to the CPUQ value. ADDR(0–7) lines are set to 0. We wish to read word B of the first element in the CPU Queue

3. WRITE EVENTCASE REG—Write the EventCase Register with the EventCase value of the next task to execute on the CPU. Actions—The EventCase Register is set equal to the value of the lines D(0–7). Actions of last state are de-asserted.

4. SETUP END uP XACTION—Setup to end transaction with CPU. Actions—DTACK (data transfer acknowledge) is asserted. Event—cs is de-asserted.

5. END uP XACTION—End transaction with CPU. Actions—DTACK (data transfer acknowledge) is de-asserted.

6. IDLE (state 1 above)

It should be understood that the foregoing description is only illustrative of the invention. Various alternatives and modifications can be devised by those skilled in the art without departing from the invention. Accordingly, the present invention is intended to embrace all such alternatives, modifications and variances which fall within the scope of the appended claims.

I claim:

1. A multitasking data processing system including a hardware-configured portion of an operating system, the combination comprising:

processor queue means configured in hardware and including a plurality of word stores, for storing in priority order, task names ready for execution;

event queue means configured in hardware and including a plurality of word stores, for storing task names that await an occurrence of an event to be placed in said processor queue means; processor means for signalling occurrence of an event; and

match logic means configured in hardware and responsive to an asynchronously signalled event from said processor means for searching said word stores in parallel in said event queue means to find a task name associated with said signalled event occurrence, and for transferring said task name to said processor queue means.

2. The multitasking data processing system recited in claim **1**, further comprising:

shift logic means for simultaneously transferring a plurality of task names from one set of word stores to another set of word stores in said processor queue means.

3. The multitasking data processing system recited in claim **2**, wherein said shift logic means includes means for simultaneously transferring said plurality of task names from word stores in a first direction along said queue or in the opposite direction along said queue, in dependence upon a generated command signal from said processor means.

4. The multitasking data processing system as recited in claim **3**, wherein said shift logic means, in response to an address of a said word store, causes said simultaneous transfer of task names to commence from said addressed word store and to include task names in additional word stores along said processor queue, starting from said addressed word store.

5. A multitasking data processing system recited in claim **1**, further comprising:

hardware event count means, including a plurality of registers, each register storing a count of tasks awaiting said asynchronously signalled event from said proces-

sor means; and

search means responsive to said asynchronously signalled event to search said registers in parallel in said hardware event count means to determine whether a task is awaiting said asynchronously signalled event, and if so, operating said match logic means to search said event queue means to find a name of a task awaiting said asynchronously signalled event.

6. A multitasking data processing system recited in claim **5**, wherein each said word store in said event queue means stores a task name and an associated event name, an occurrence of said named event causing said named task to be transferred to said processor queue means.

7. A multitasking data processing system recited in claim **6**, wherein task names in said event queue means are stored in a priority order assigned to each task, and a parallel search of said event queue means by said match logic means causes a readout of all task names associated with a signalled event, in parallel and in priority order.

8. A multitasking data processing system recited in claim **5**, wherein said search means further determines, for a signalled event, if more events have occurred than there are tasks awaiting such event's occurrence.

9. A multitasking data processing system as recited in claim **1** further comprising:

hardware delay queue means including a plurality of word stores, each word store storing a delay interval value and an event to be signalled by said processor means upon occurrence of said delay interval value;

timer means for signalling delay interval values;

means responsive to a delay interval value signalled by said timer means for searching word stores in said delay queue means, in parallel, to find an event to be signalled upon a signalling of a delay interval value and to transmit said signalled event to said match logic means.

10. A multitasking data processing system as recited in claim **1** wherein each said task name is represented by a unique value, said value also indicating said named task's priority among all tasks.

11. A hardware-configured operating system kernel, comprising:

a set of command and status hardware registers for receiving both addresses and data from a connected data processing system and for passing data to said data processing system;

a CPU queue including a plurality of hardware word stores for storing a queue of task names, said queue of task names organized in priority order of said named tasks, all said tasks being ready for execution;

a hardware-configured event queue including a plurality of word stores for storing a queue of task names in priority order, each said task name stored in association with an event name upon whose occurrence, said associated task name will be ready for execution; and

a queue state machine responsive to asynchronously occurring event data in said command and status registers, to cause a parallel search of said event queue to find all task names associated with said asynchronously occurring event data, and to enable a transfer of said task names to said CPU queue.

12. The kernel as recited in claim **11**, wherein said queue state machine causes said task names to be stored in said CPU queue in task name priority order.

13. The kernel as recited in claim **12**, wherein each said task name is a unique value, said value indicative of said named task's priority.

14. The kernel as recited in claim **13**, further comprising:

a delay queue including a plurality of word stores for storing a queue of delay interval values and associated event names, said event names ordered in said delay queue by increasing value of associated delay interval values, said queue state machine responsive to a delay interval value manifestation to cause an associated event name to be signalled and task name transfers from said event queue to be enabled.

15. The kernel as recited in claim **14**, further comprising:

address means connected to each word store in each said queue for enabling data to be selectively read from or written into any said word store.

16. The kernel as recited in claim **15**, further comprising:

bus means connecting each said word store in each said queue for enabling simultaneous data transfers between selected word stores.

17. The kernel as recited in claim **16**, further comprising:

a plurality of event registers, one for each of a plurality of events, each said event register indicating a count of tasks awaiting occurrence of an event, said queue state machine responsive to a signalled event to search all said event registers in parallel to determine if any tasks were awaiting occurrence of said signalled event, and if so, to cause transfer of a highest priority task name associated with said event in said event queue.

* * * * *

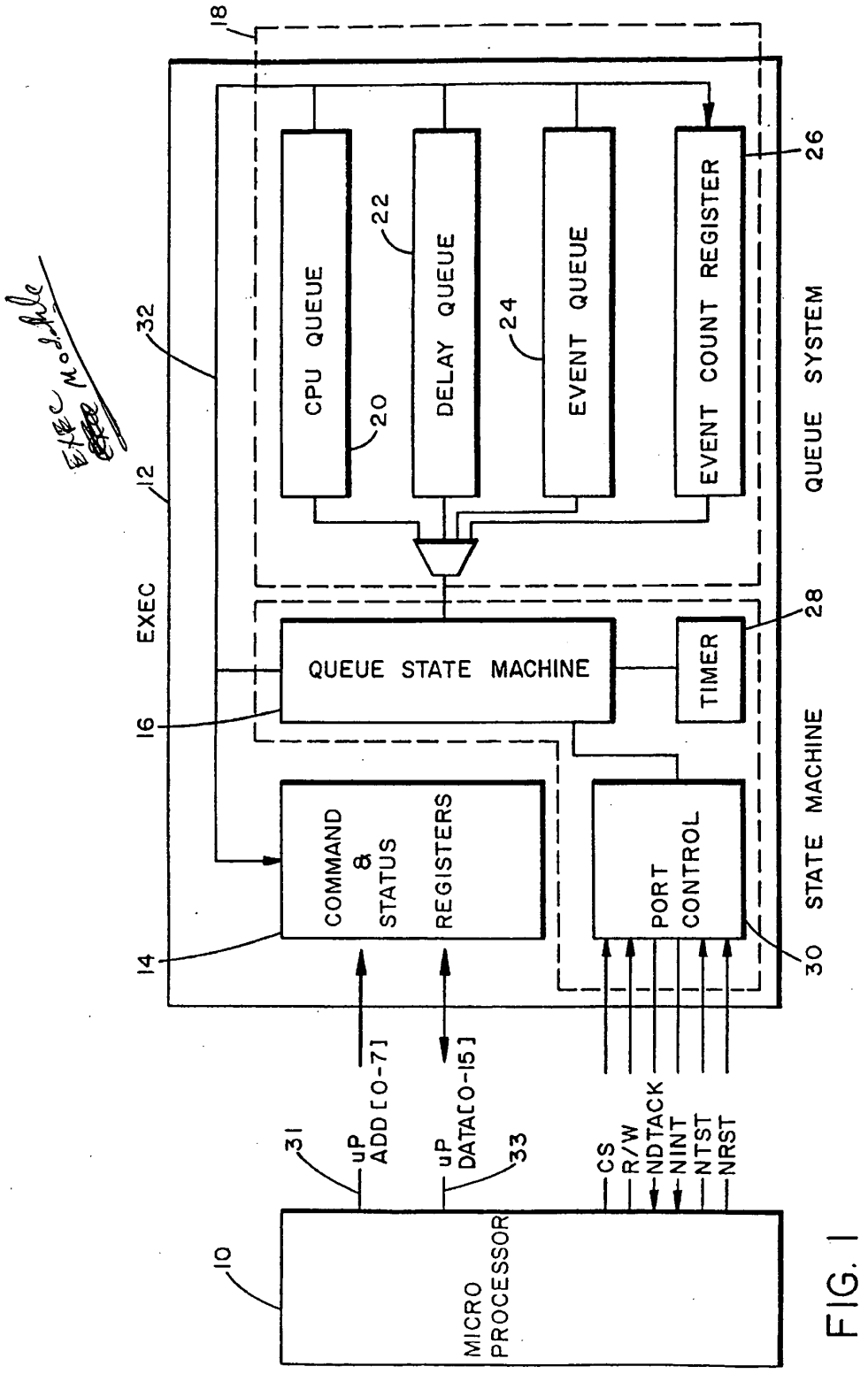
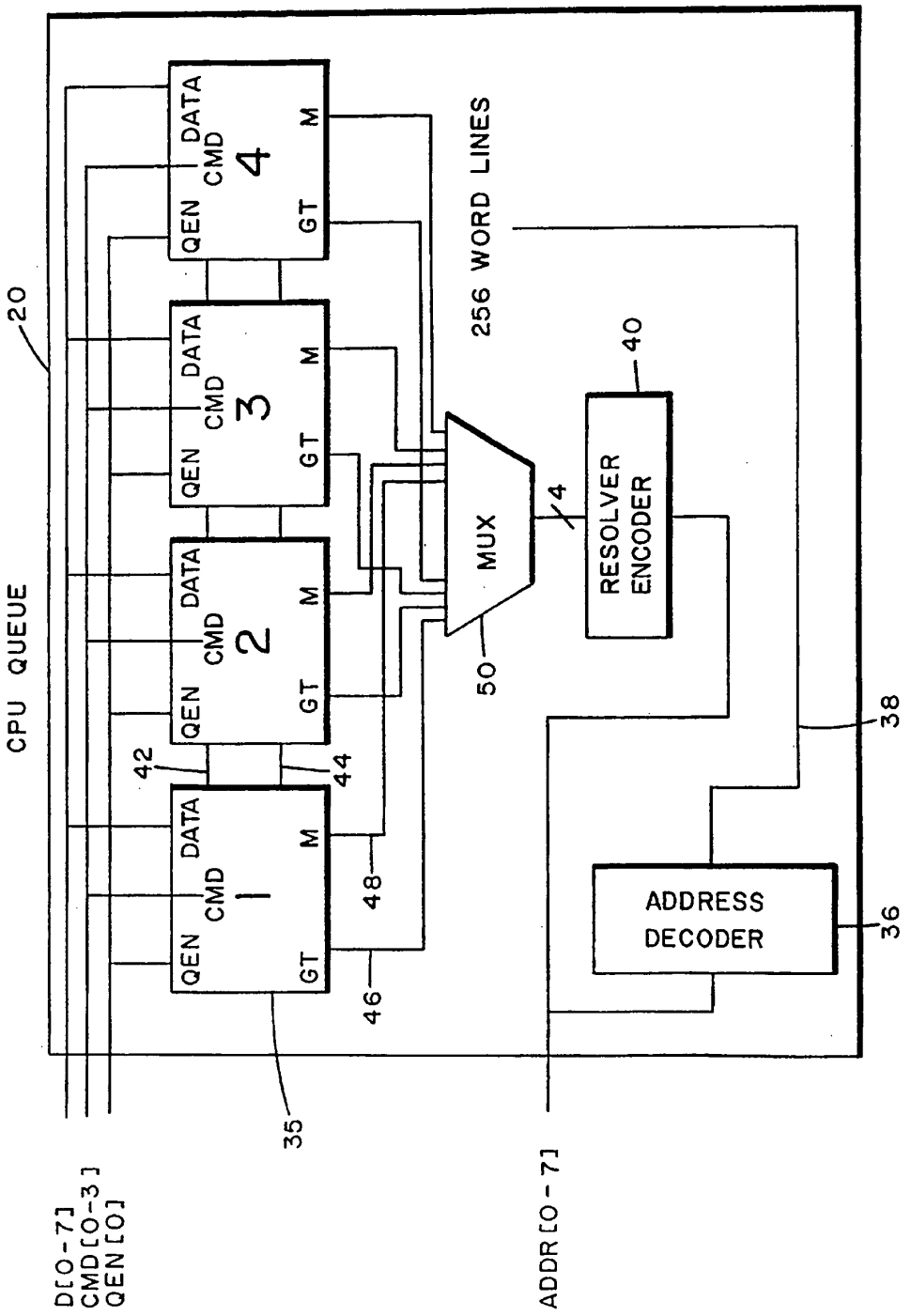

18
CPU QUEUE
20
DELAY QUEUE
22
EVENT QUEUE
24
EVENT COUNT REGISTER
26
QUEUE SYSTEM
32
12
EXEC
QUEUE STATE MACHINE
TIMER
28
16
COMMAND & STATUS REGISTERS
PORT CONTROL
30
STATE MACHINE
14
uP ADD[0-7]
uP DATA[0-15]
31
33
CS
R/W
NDTACK
NINT
NTST
NRST
MICRO PROCESSOR
10

FIG. 1

FIG. 2


20
CPU QUEUE
DATA
CMD
QEN
M
GT
4
3
2
1
256 WORD LINES
MUX
50
4
RESOLVER ENCODER
40
38
ADDRESS DECODER
36
42
44
46
48
35
D[0-7]
CMD[0-3]
QEN[0]
ADDR[0-7]

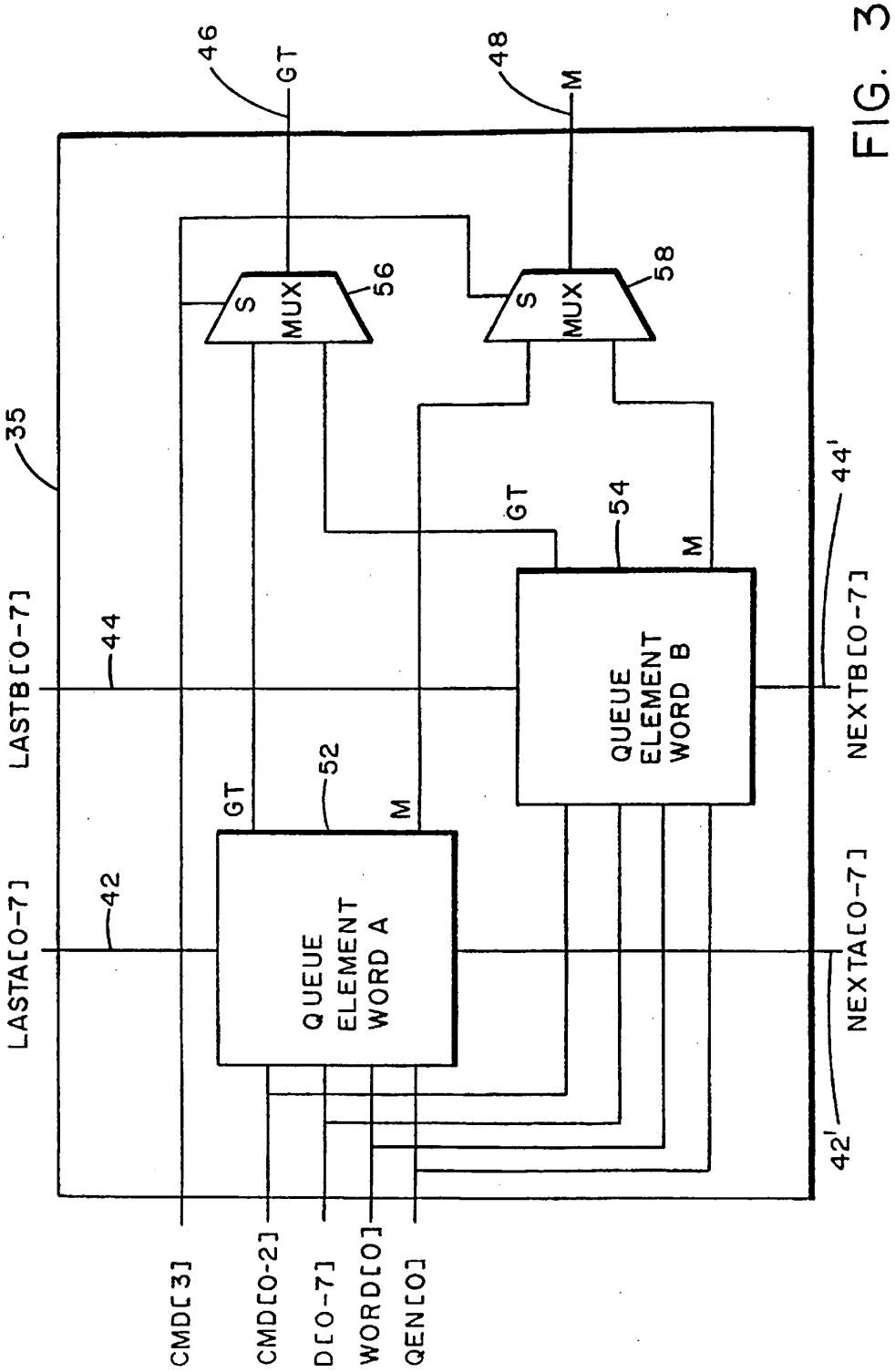

46
GT
48
M
S
MUX
56
S
MUX
58
35
GT
54
M
44'
LASTB[0-7]
44
QUEUE
ELEMENT
WORD B
NEXTB[0-7]
GT
52
M
LASTA[0-7]
42
QUEUE
ELEMENT
WORD A
NEXTA[0-7]
42'
CMD[3]
CMD[0-2]
D[0-7]
WORD[0]
QEN[0]

FIG. 3

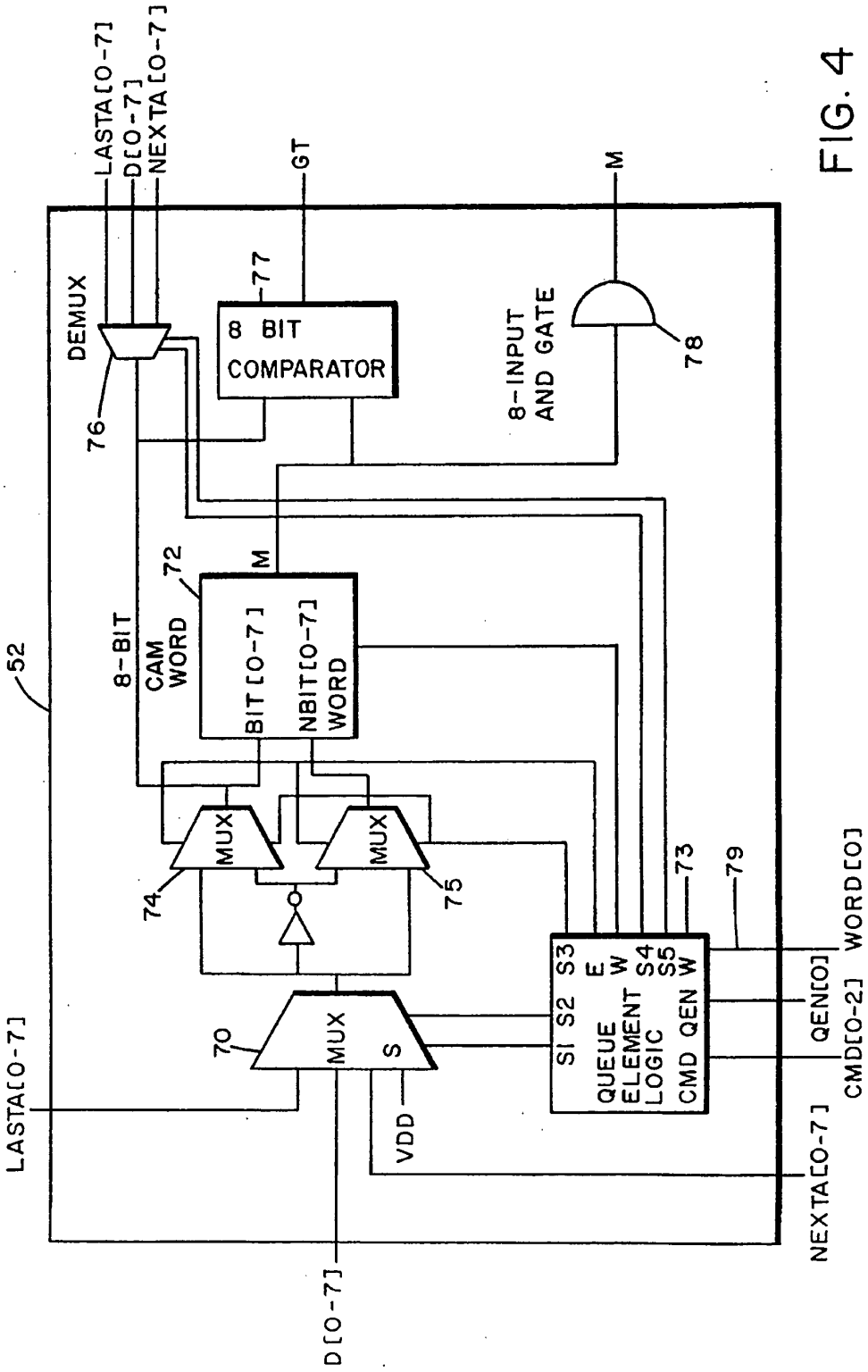

LASTA[0-7]
D[0-7]
NEXTA[0-7]
GT
M
DEMUX
76
77
8 BIT COMPARATOR
8-INPUT AND GATE
78
72
8-BIT CAM WORD
BIT[0-7]
NBIT[0-7] WORD
52
MUX
MUX
74
75
73
79
70
MUX
S
SI S2 S3 E W S4 S5
QUEUE ELEMENT LOGIC
CMD QEN W
QEN[0]
WORD[0]
CMD[0-2]
LASTA[0-7]
VDD
NEXTA[0-7]
D[0-7]

FIG. 4

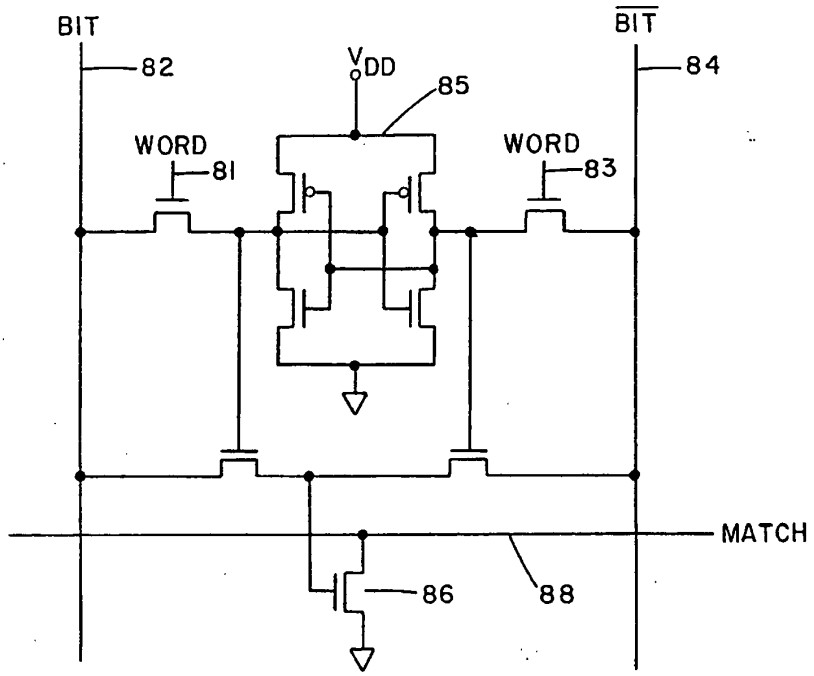

BIT
82
VDD
85
BIT
84
WORD
81
WORD
83
MATCH
86
88

FIG. 5

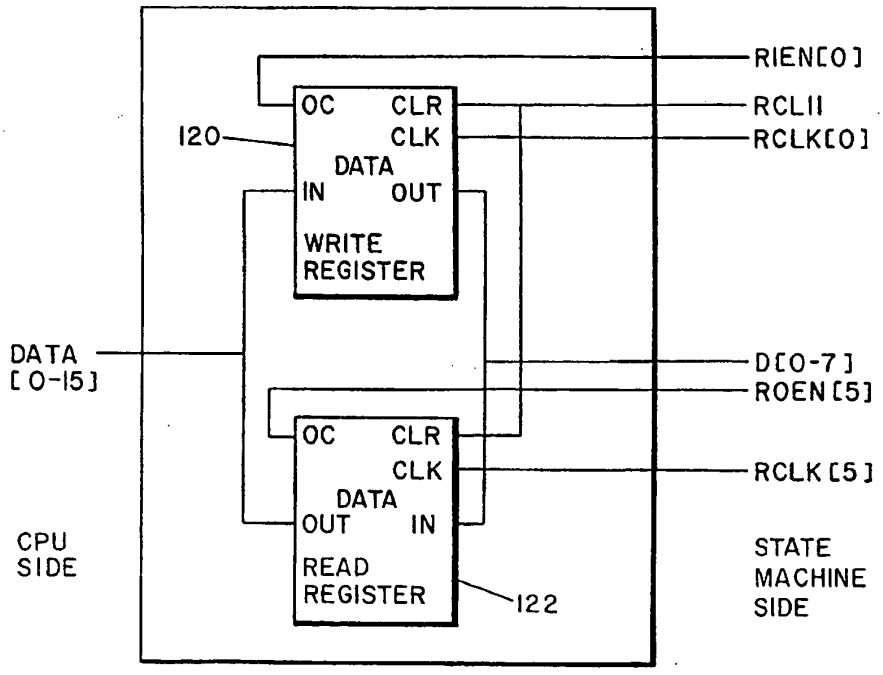

RIEN[0]
OC
CLR
RCLII
120
CLK
RCLK[0]
DATA
IN
OUT
WRITE
REGISTER
DATA
[0-15]
D[0-7]
ROEN[5]
OC
CLR
CLK
RCLK[5]
DATA
OUT
IN
CPU
SIDE
READ
REGISTER
122
STATE
MACHINE
SIDE

FIG. 8

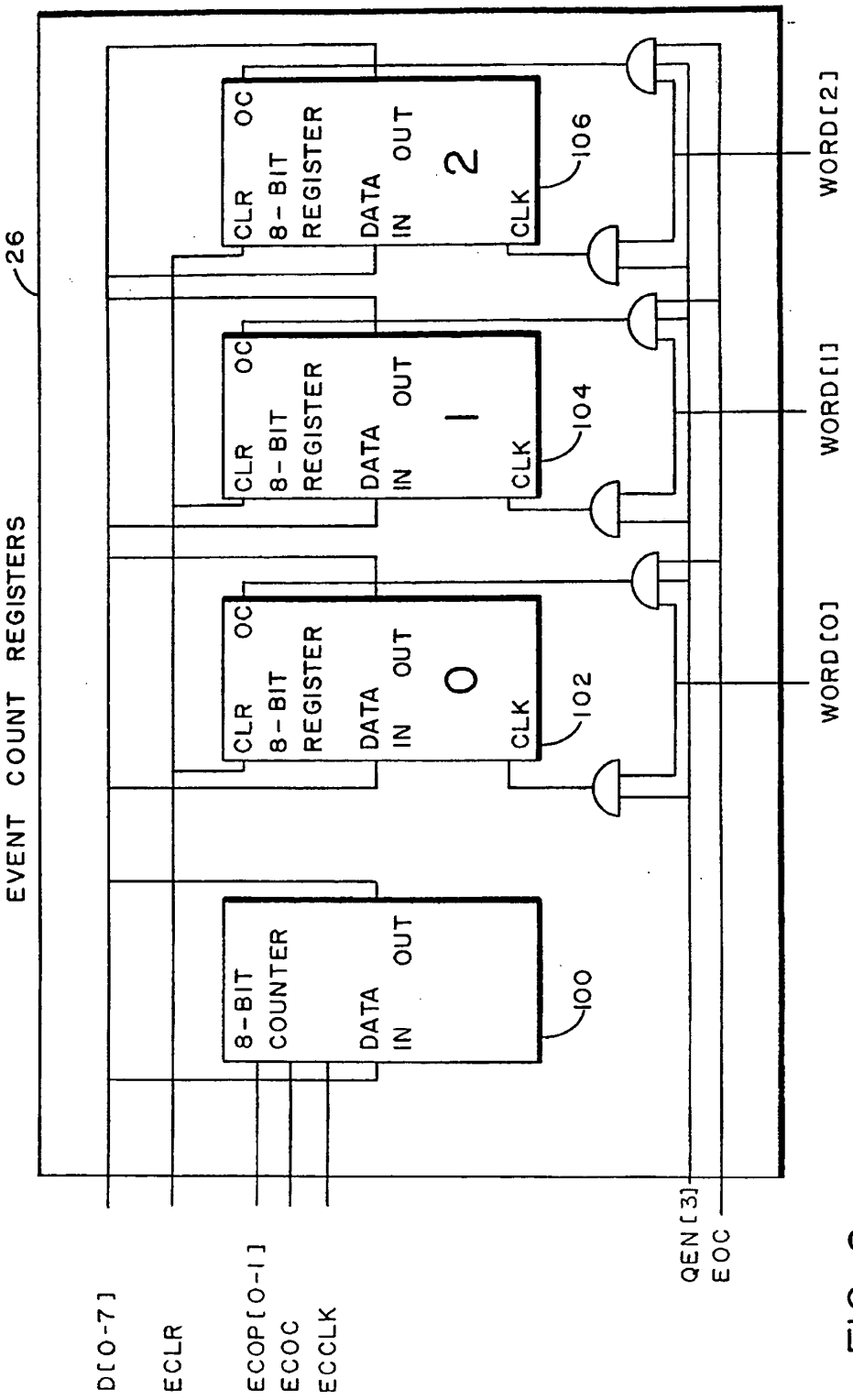

26
EVENT COUNT REGISTERS
8-BIT COUNTER
DATA IN OUT
100
8-BIT REGISTER
CLR OC
DATA IN OUT
CLK
0
102
1
104
2
106
WORD[0]
WORD[1]
WORD[2]
QEN[3]
EOC
D[0-7]
ECLR
ECOP[0-1]
ECOC
ECCLK

FIG. 6

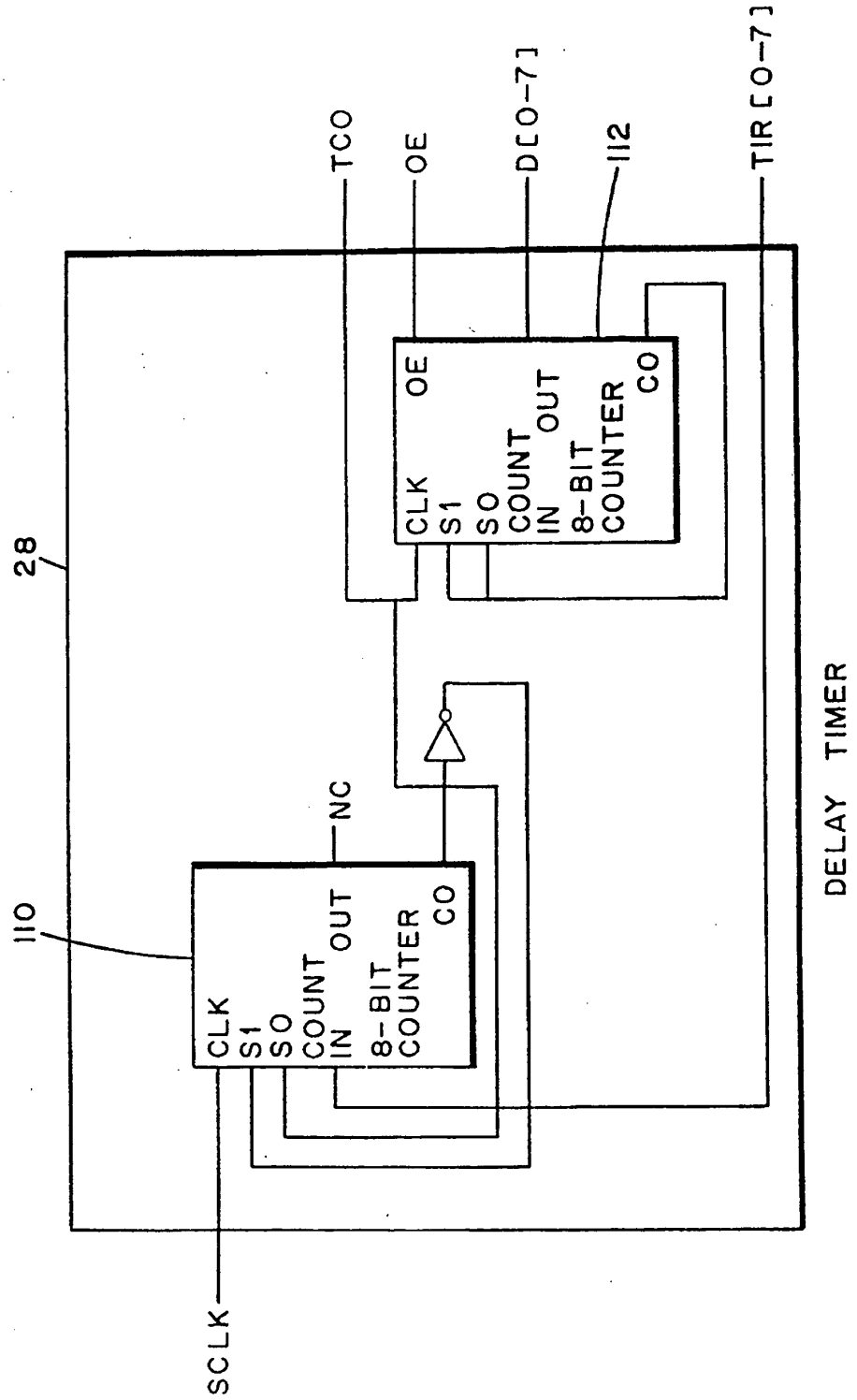

TCO
OE
D[0-7]
112
TIR[0-7]
28
CLK
S1
SO
COUNT IN
8-BIT COUNTER
OUT
CO
NC
110
SCLK
DELAY TIMER

FIG. 7

US005790851A

United States Patent [19]

Frank et al.

[11] Patent Number: 5,790,851

[45] Date of Patent: Aug. 4, 1998

[54] **METHOD OF SEQUENCING LOCK CALL REQUESTS TO AN O/S TO AVOID SPINLOCK CONTENTION WITHIN A MULTI-PROCESSOR ENVIRONMENT**

[75] Inventors: **Richard Frank**, Groton, Mass.; **Gopolan Arun**; **Richard Anderson**, both of Nashua, N.H.; **Stephen Klein**, Hollis, N.H.

[73] Assignee: **Oracle Corporation**, Redwood Shores, Calif.

[21] Appl. No.: **843,332**

[22] Filed: **Apr. 15, 1997**

[51] **Int. Cl.**[6] **G06F 9/40**

[52] **U.S. Cl.** **395/674**; 395/670

[58] **Field of Search** 395/674, 670

[56] **References Cited**

U.S. PATENT DOCUMENTS

5,251,318	10/1993	Nitta et al.	395/725
5,455,944	10/1995	Haderle et al.	395/600
5,623,659	4/1997	Shi et al.	395/608

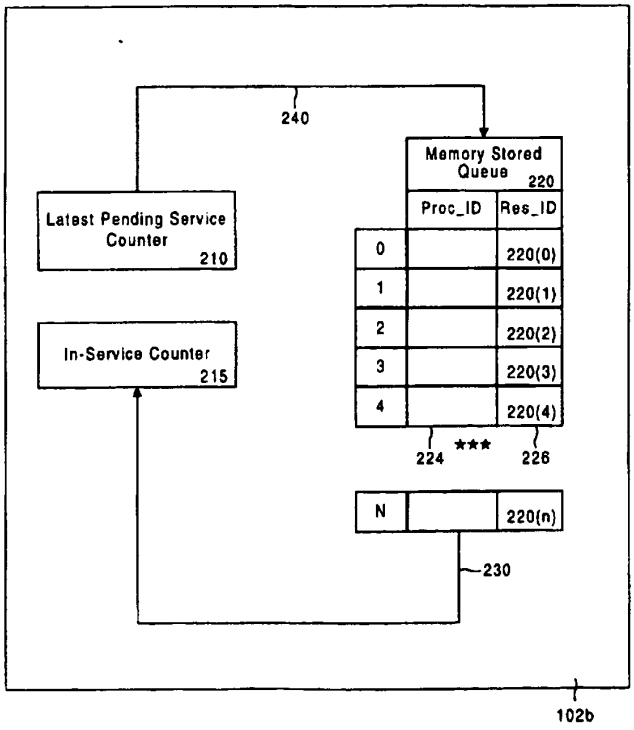
Primary Examiner—Lucien U. Toplu

Attorney, Agent, or Firm—Wagner, Murabito & Hao

[57] **ABSTRACT**

An arbitration procedure allowing processes and their associated processors to perform useful work while they have pending service requests for access to shared resources within a multi-processor system environment. The arbitration procedure of the present invention is implemented within a multi-processor system (e.g., a symmetric multi-processor system) wherein multiple processes can simultaneously request "locks" which control access to shared resources such that access to these shared resources are globally synchronized among the many processes. Rather than assigning arbitration to the operating system, the present invention provides an arbitration procedure that is application-specific. This arbitration process provides a reservation mechanism for contending processes such that any given process only requests a lock call to the operating system when a lock is available for that process, thereby avoiding spinlock by the operating system. During the period between a lock request and a lock grant, a respective process is allowed to perform other useful work that does not need access to the shared resource. Alternatively during this period, the processor executing the respective process can execute another process that performs useful work that does not need the shared resource. Each process requesting a lock grant is informed of the expected delay period, placed on a reservation queue, and assigned a reservation identifier. After releasing the lock, the process uses the reservation queue to locate the next pending process to receive the lock.

21 Claims, 11 Drawing Sheets



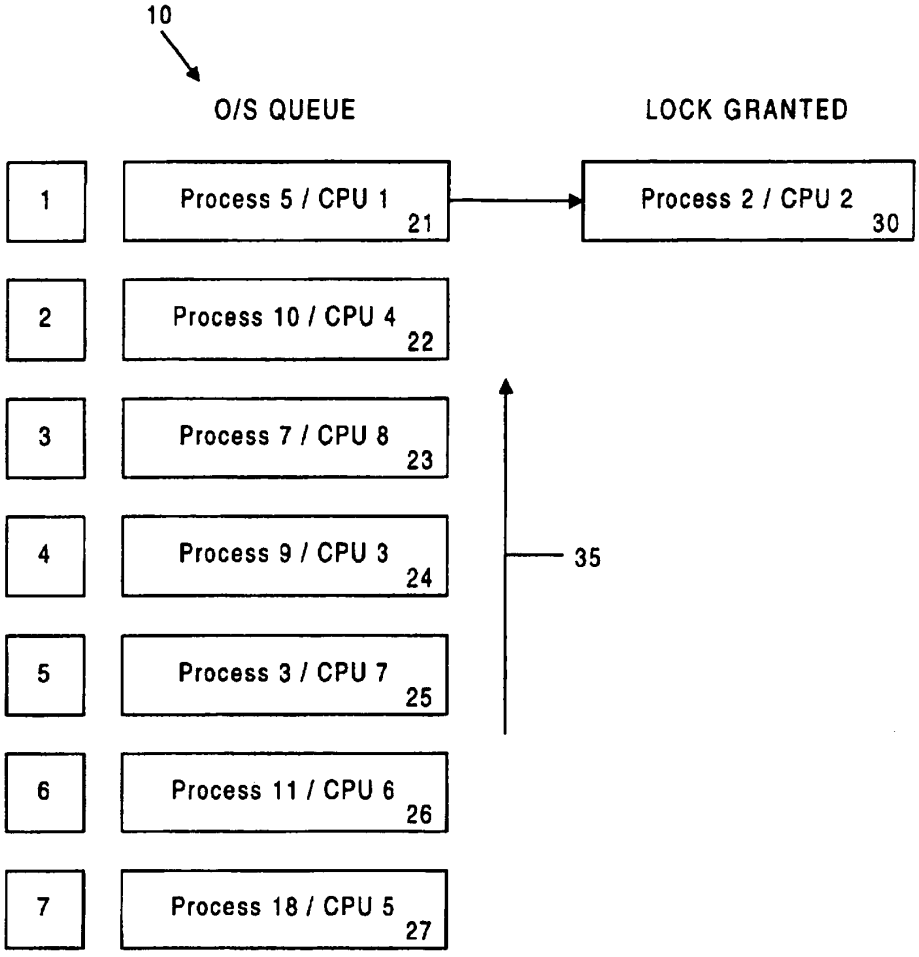


FIG. 1
(Prior Art)

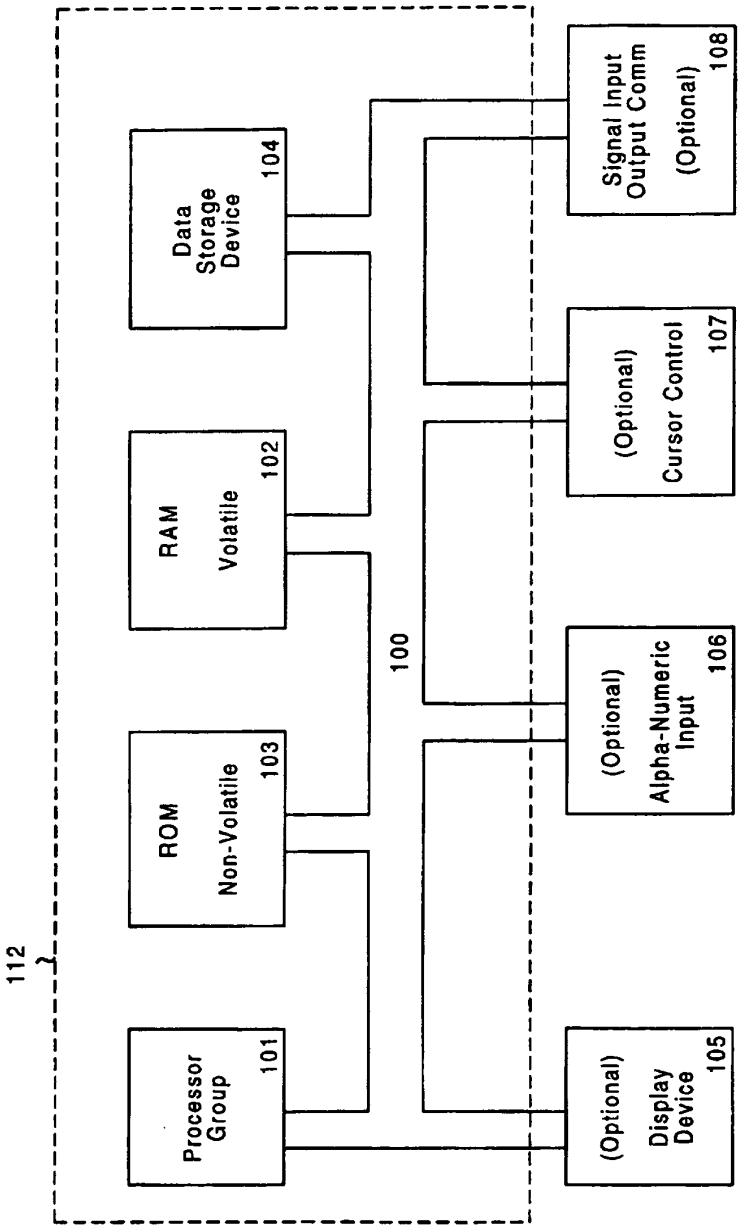

112
Processor Group 101
ROM Non-Volatile 103
RAM Volatile 102
Data Storage Device 104
100
(Optional) Display Device 105
(Optional) Alpha-Numeric Input 106
(Optional) Cursor Control 107
Signal Input Output Comm (Optional) 108

FIG. 2A

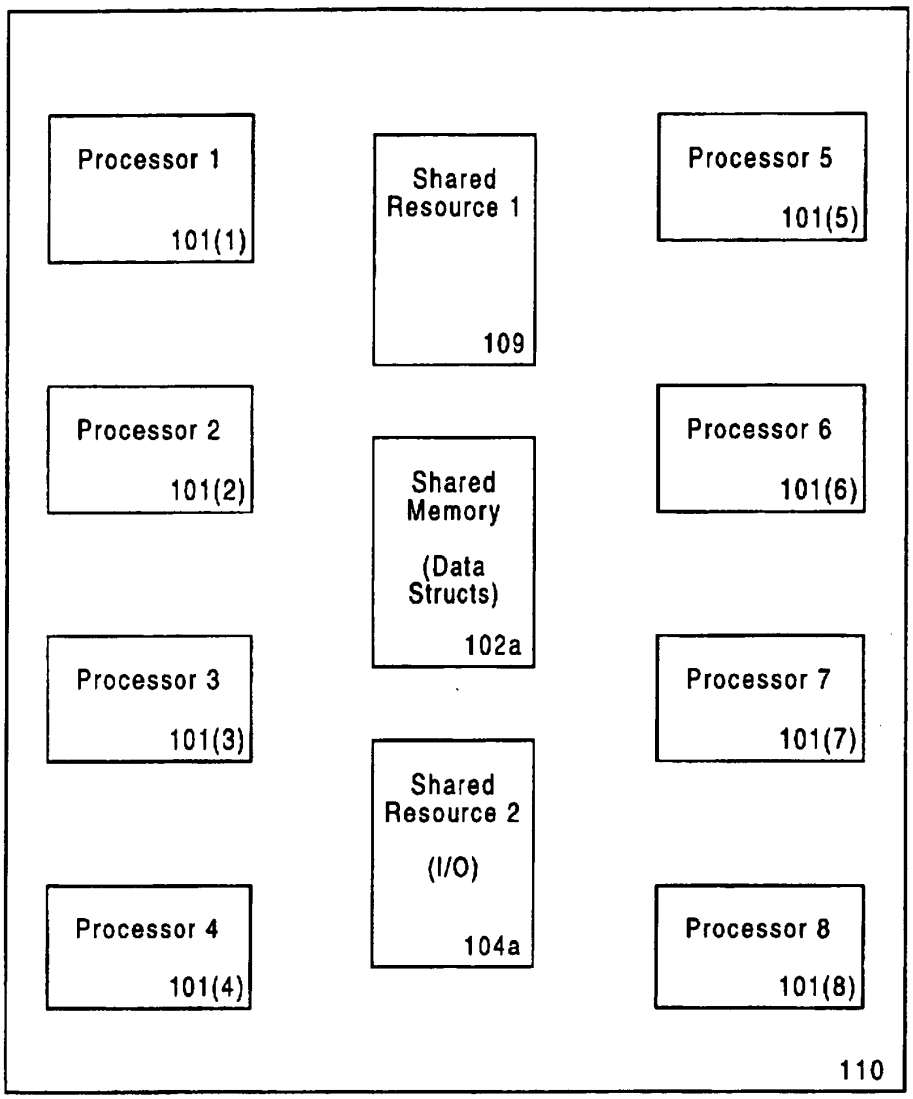

Processor 1
101(1)
Shared Resource 1
109
Processor 5
101(5)
Processor 2
101(2)
Shared Memory
(Data Structs)
102a
Processor 6
101(6)
Processor 3
101(3)
Shared Resource 2
(I/O)
104a
Processor 7
101(7)
Processor 4
101(4)
Processor 8
101(8)
110

FIG. 2B

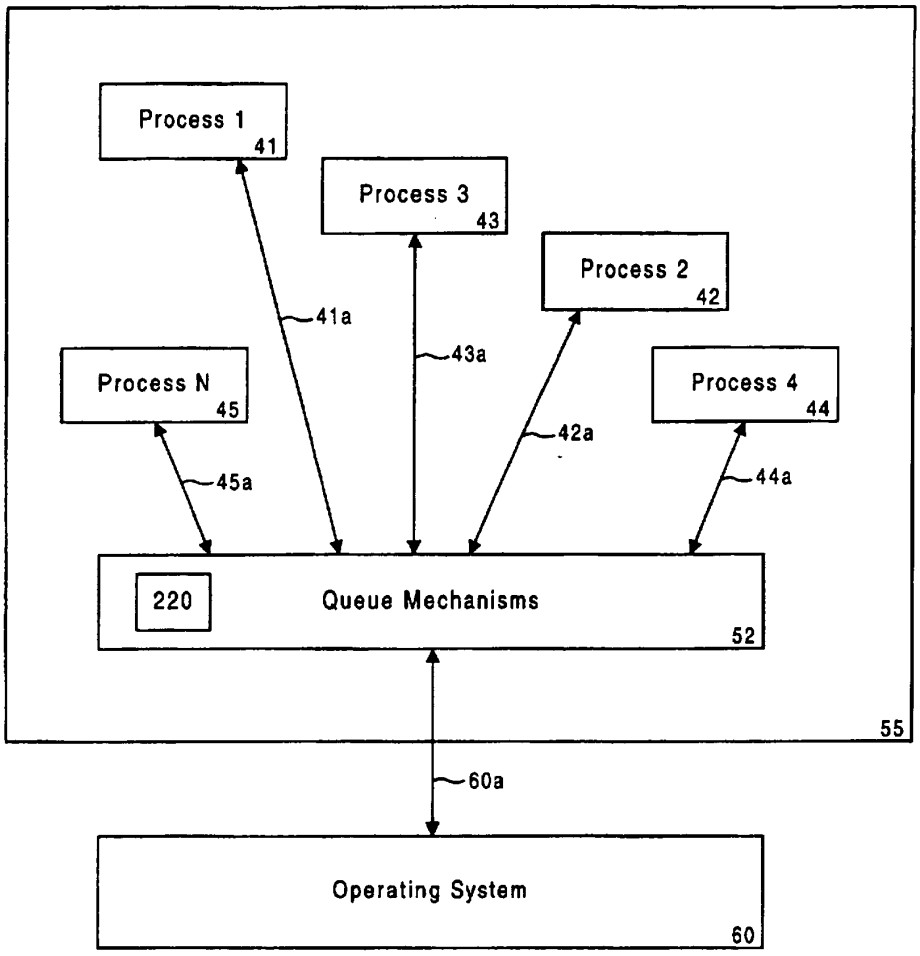

Process 1
41
Process 3
43
Process 2
42
41a
43a
Process N
45
Process 4
44
42a
45a
44a
220
Queue Mechanisms
52
55
60a
Operating System
60

FIG. 2C

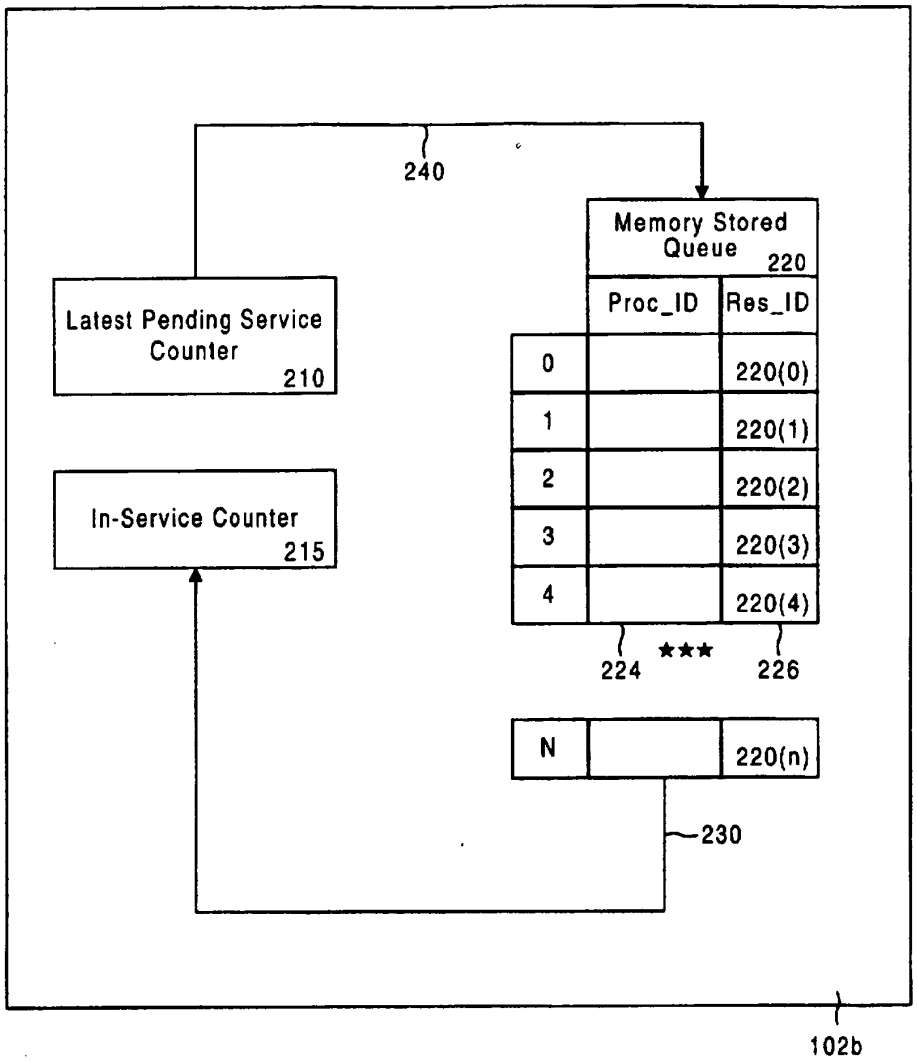

240
Memory Stored Queue
220
Proc_ID
Res_ID
Latest Pending Service Counter
210
In-Service Counter
215
0
1
2
3
4
N
220(0)
220(1)
220(2)
220(3)
220(4)
220(n)
224
226
230
102b

FIG. 3

400

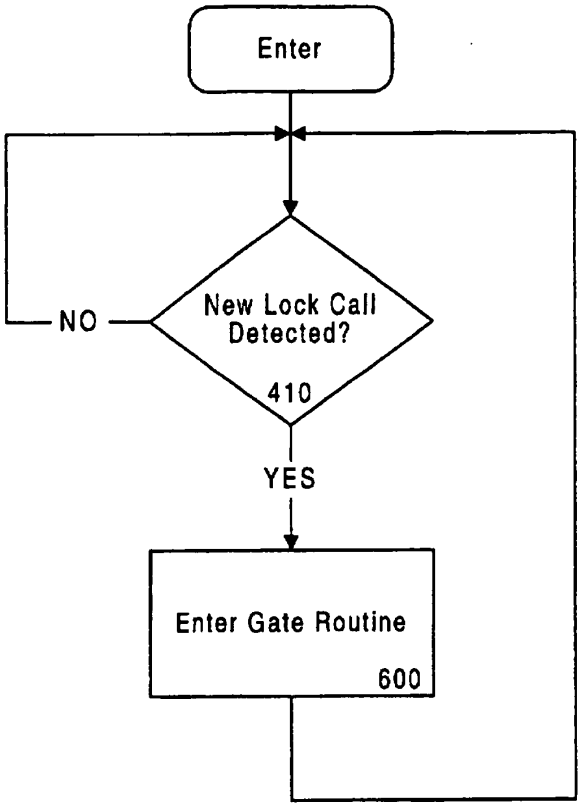

Enter
New Lock Call Detected?
410
NO
YES
Enter Gate Routine
600

FIG. 4

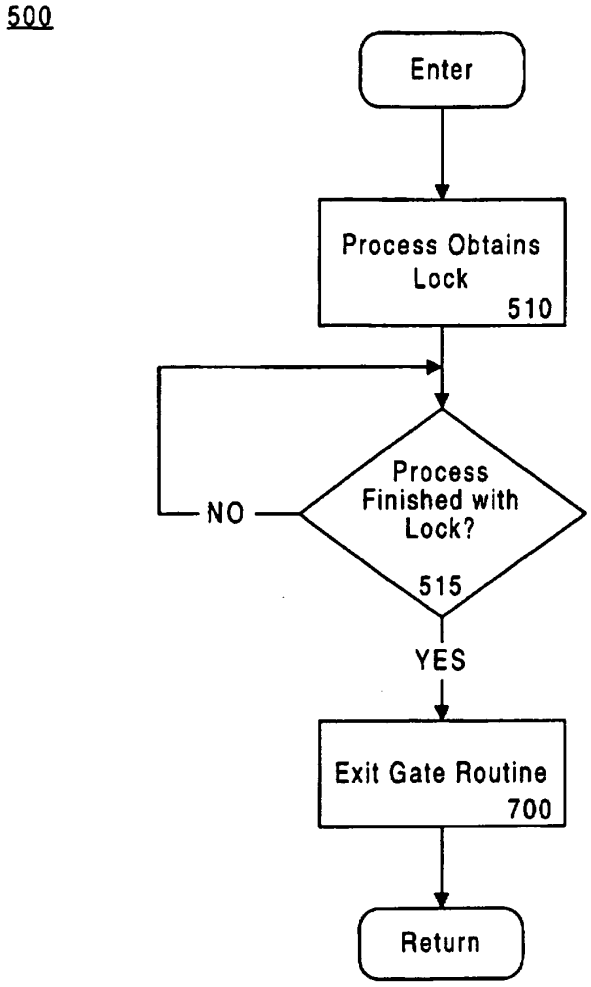

500
Enter
Process Obtains Lock
510
Process Finished with Lock?
515
NO
YES
Exit Gate Routine
700
Return

FIG. 5

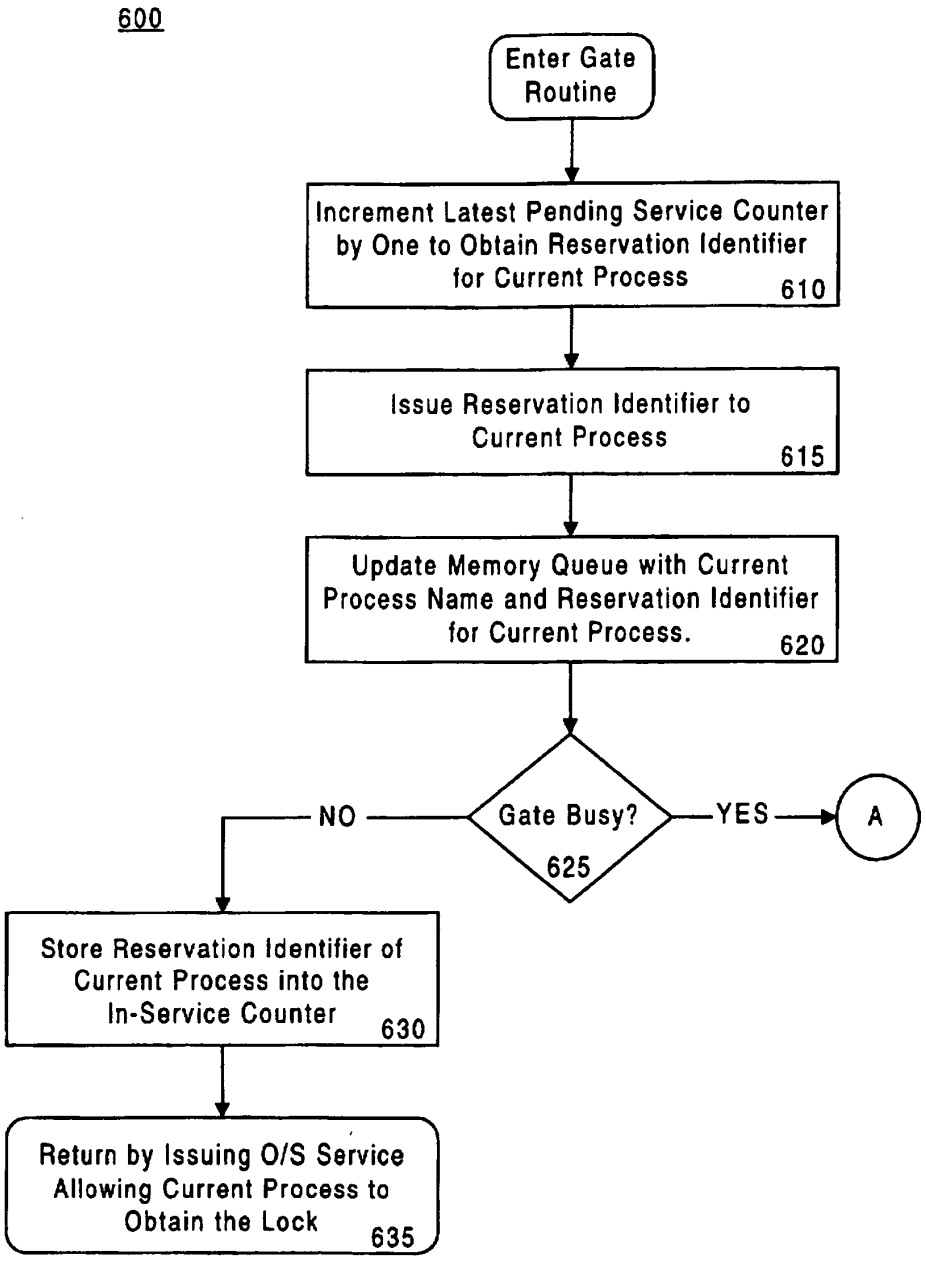

600
Enter Gate Routine
Increment Latest Pending Service Counter by One to Obtain Reservation Identifier for Current Process
610
Issue Reservation Identifier to Current Process
615
Update Memory Queue with Current Process Name and Reservation Identifier for Current Process.
620
Gate Busy?
625
NO
YES
A
Store Reservation Identifier of Current Process into the In-Service Counter
630
Return by Issuing O/S Service Allowing Current Process to Obtain the Lock
635

FIG. 6A

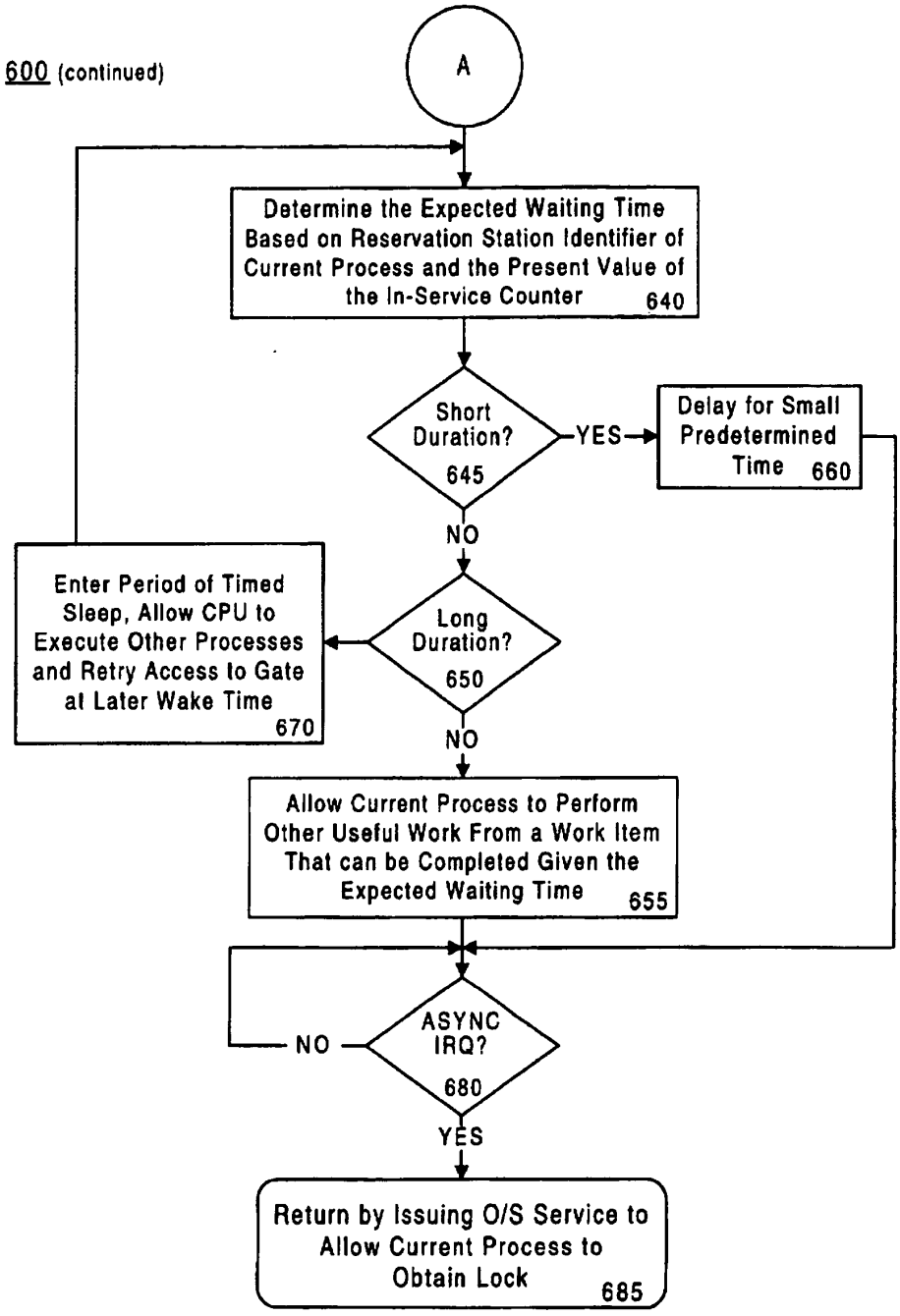

600 (continued)
A
Determine the Expected Waiting Time Based on Reservation Station Identifier of Current Process and the Present Value of the In-Service Counter 640
Short Duration? 645
YES
Delay for Small Predetermined Time 660
NO
Long Duration? 650
Enter Period of Timed Sleep, Allow CPU to Execute Other Processes and Retry Access to Gate at Later Wake Time 670
NO
Allow Current Process to Perform Other Useful Work From a Work Item That can be Completed Given the Expected Waiting Time 655
ASYNC IRQ? 680
NO
YES
Return by Issuing O/S Service to Allow Current Process to Obtain Lock 685

FIG. 6B

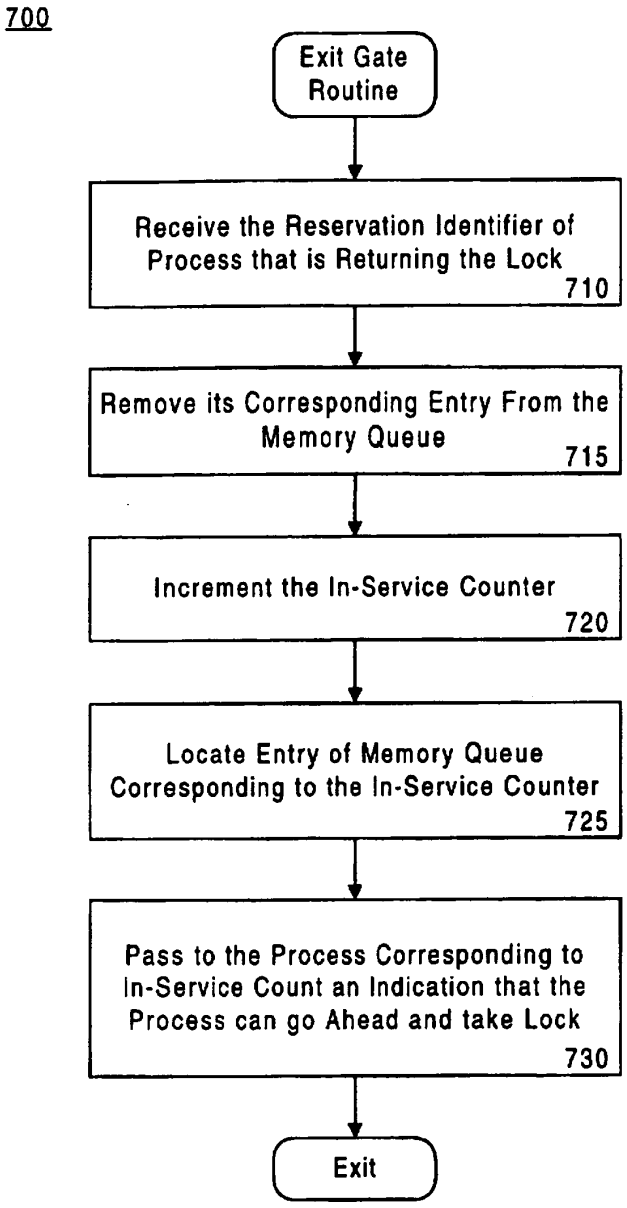

700
Exit Gate Routine
Receive the Reservation Identifier of Process that is Returning the Lock
710
Remove its Corresponding Entry From the Memory Queue
715
Increment the In-Service Counter
720
Locate Entry of Memory Queue Corresponding to the In-Service Counter
725
Pass to the Process Corresponding to In-Service Count an Indication that the Process can go Ahead and take Lock
730
Exit

FIG. 7

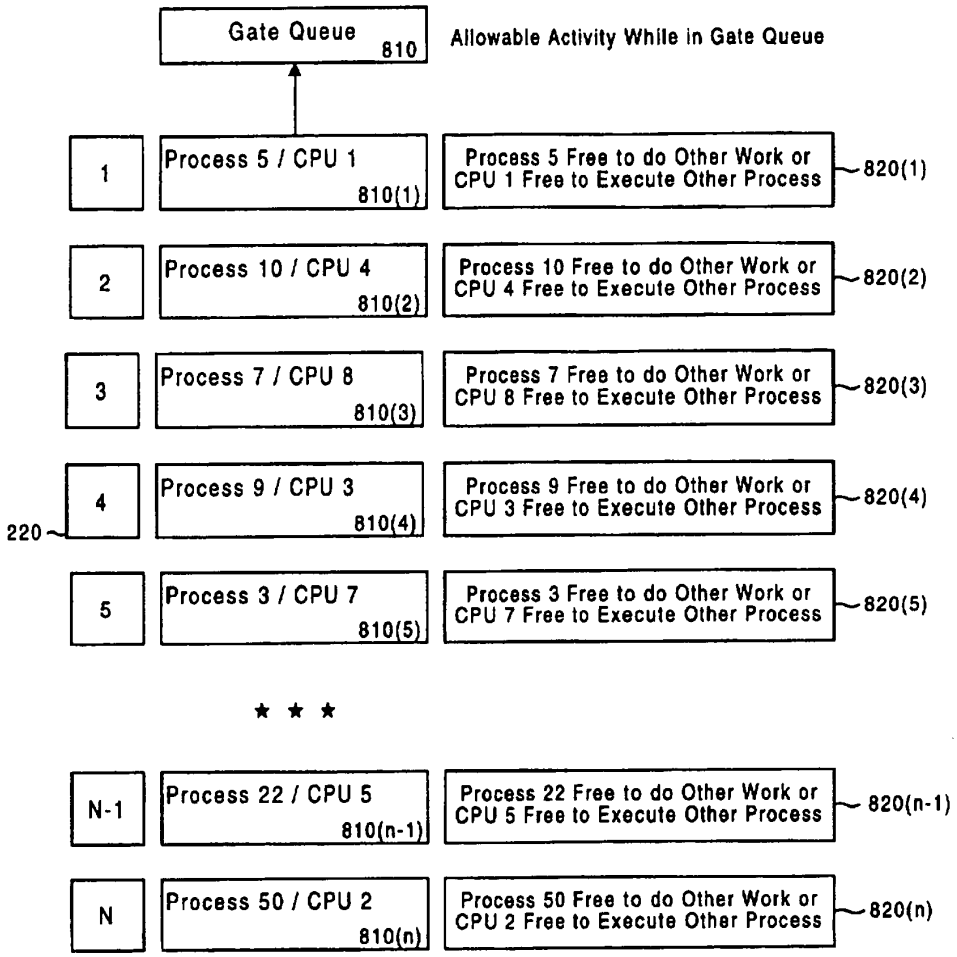

Gate Queue
810
Allowable Activity While in Gate Queue
1
Process 5 / CPU 1
810(1)
Process 5 Free to do Other Work or CPU 1 Free to Execute Other Process
820(1)
2
Process 10 / CPU 4
810(2)
Process 10 Free to do Other Work or CPU 4 Free to Execute Other Process
820(2)
3
Process 7 / CPU 8
810(3)
Process 7 Free to do Other Work or CPU 8 Free to Execute Other Process
820(3)
4
Process 9 / CPU 3
810(4)
Process 9 Free to do Other Work or CPU 3 Free to Execute Other Process
820(4)
220
5
Process 3 / CPU 7
810(5)
Process 3 Free to do Other Work or CPU 7 Free to Execute Other Process
820(5)
★ ★ ★
N-1
Process 22 / CPU 5
810(n-1)
Process 22 Free to do Other Work or CPU 5 Free to Execute Other Process
820(n-1)
N
Process 50 / CPU 2
810(n)
Process 50 Free to do Other Work or CPU 2 Free to Execute Other Process
820(n)

FIG. 8

METHOD OF SEQUENCING LOCK CALL REQUESTS TO AN O/S TO AVOID SPINLOCK CONTENTION WITHIN A MULTI-PROCESSOR ENVIRONMENT

BACKGROUND OF THE INVENTION

(1) Field of the Invention

The present invention relates to the field of computer systems. More specifically, the present invention relates to the field of arbitration for system resources between processors in a multi-processor environment.

(2) Prior Art

Multi-processor computer systems are designed with multiple processors that share certain resources, e.g., shared memory, shared data structures, shared input/output devices, etc. One such multi-processor system is the symmetric multi-processor system or SMP. In a multi-processor system environment, different software implemented processes (also called procedures) can be assigned to be executed by different processors of the system. To the extent that these different processes do not require access to shared resources, a significant amount of parallel processing can be performed by the multi-processor system, and the addition of more processors may increase overall data throughput. Efficient or effective use of parallel processing greatly decreases the amount of processing time required to perform certain tasks, for instance, tasks performed by a relational database management system (RDBMS). Therefore, many RDBMS systems employ multi-processor systems to efficiently perform database tasks.

System scaling refers to increasing system performance and processing efficiency by the addition of more processors within a multi-processor system. However, there are cases in which the addition of more processors within a multi-processor system can actually decrease overall system performance and processing efficiency. In one such case, competition or "contention" between processors for the same shared resources that are available within the multi-processor system renders some processors inactive. Typically, the contention between processors in the multi-processor system is represented as contention between different processes running on these processors. Therefore, contention is generally expressed as being process specific, or as existing between processes, but nevertheless has enormous ramifications for the processors themselves.

For instance, when shared system resources are needed by a number of processes at the same time, prior art contention handling routines that are built into an operating system (O/S) determine which process is first allowed access to the shared resource. The other processes that request access, but are not selected by the operating system, are then queued by the operating system, theoretically providing "fairness" for the resources since they are "shared." The operating system is typically informed of a process' need for shared resource by receiving a lock call from the process. This lock call specifies that a "lock" is desired over a particular shared resource by the process. The "lock" is an authorization to use the shared resource and is given to the process with the highest contention priority. In this fashion, global access synchronization is maintained between the various processes over the shared resource.

However, within prior art lock contention arbitration (or spinlock contention arbitration) procedures that are built into operation systems, those processes that execute a lock call, but are denied access to the lock, are forced into a spin routine (spinlock) whereby their associated processors are held up during a waiting period for the lock (also called "busy" period). During this waiting period, no useful work can be performed by the processor (e.g., the processor is prevented from executing the process, or any processes at all, nor can the processor receive or service an interrupt). Generally, the only function the processor performs during this waiting period is to periodically check if the lock has been granted to it by the operating system. During spinlock, the processor is not even informed of the expected length of the delay period until lock grant is to take place. In these prior art operating systems, very high levels of spinlock contention, e.g., on the order of multiple processors of compute cycles, may occur for specific application work loads and processor configurations.

The above is particularly true for asynchronous resource requests where the prior art spinlock contention service routines spin creating a bottleneck which reduces SMP scaling as the processes cannot continue working. As shown below, the practice of placing multiple processors in spinlock while waiting for their shared resources dramatically reduces the overall processing efficiency of any multi-processor system.

In FIG. 1, a time snap-shot is shown (e.g., t=x) of an exemplary prior art queue **10** implemented by an operating system, with respect to a multi-processor system having eight processors (referred to as CPU 1 through CPU 8). At this time, process 2 of CPU2 currently is granted a lock corresponding to an arbitrary shared resource. Within queue **10**, slots **21–27** are held by processes **5**, **10**, **7**, **9**, **3**, **11**, and **18** of CPUs **1**, **4**, **8**, **3**, **7**, **6**, and **5**, respectively. Processors in queue **10** are held in spinlock waiting for the same lock. Priority ordering for access to this lock is represented by the processes' positions (**1–7**) within queue **10**. In this state, no processors (aside from CPU 2) are performing useful work because the operating system has each process in a spinlock whereby they are merely waiting for their next turn to access the shared resource lock. Only CPU **2** is allowed to perform useful work. In this inefficient model, the addition of more processors to the multi-processor system will not significantly increase processing efficiency of the overall system. This is the case because it is more than likely that the additional processors will be tied up in spinlock over vital shared resources rather than performing useful work. Therefore, spinlock contention between multiple processors over shared resources is a serious constraint on the degree of system scaling obtainable within a multi-processor system. Another drawback of this approach is that the size of the queue **10** (slots **21–27**) is directly related to the number of processors in the multi-processor system because each processor is held in spinlock while in the queue and there are only 8 processors.

To date, the practiced solution of the application designer has been to carefully design an application to insure minimal use of system resources that require spinlock synchronization, and subsequently accept the resulting multi-processor scaling achieved by the application, however low that may be. This solution is unacceptable for at least two reasons. First, this approach forces application designers to tolerate low degrees of multi-processor scaling. Second, in many cases it is difficult to predict which system resources will create bad spinlock contention during execution because the application designer does not know, a priori, (1) over which processor a typical process will execute in a multi-processor system nor (2) which processes the other processors are executing at any given time. As such, it is very difficult to accurately design an application to insure minimal use of system resources that require spinlock syn-

chronization when the designer does not know the precise timing of such demands and which processors will be making them.

Accordingly, it would be advantageous to provide a multi-processor system that avoids the spinlock contention problems described above. Further, it would be advantageous to provide a multi-processor system that allows processes and processors that are contending for shared resources to perform useful work while their resource requests are pending. The present invention provides such a multi-processor system which uses an advantageous lock contention arbitration procedure that avoids the above discussed shortcomings.

SUMMARY OF THE INVENTION

An arbitration procedure is described allowing processes and their associated processors to perform useful work while they have pending service requests for access to shared resources within a multi-processor system environment. The arbitration procedure of the present invention is implemented within a multi-processor system (e.g., a symmetric multi-processor system) wherein multiple processes can simultaneously request access to "locks" which synchronize access to shared resources such that access to these shared resources are globally synchronized among the many processes. Rather than assigning arbitration to the operating system, the present invention provides an arbitration procedure that is application specific allowing an application to retain control of its processor while waiting for a lock grant. The present invention sits on top of the operating system and requests locks of the operating system only when they are known to be available.

This arbitration process provides a reservation mechanism for contending processes such that any given process only requests a lock call to the operating system when a lock is available for that process, thereby avoiding spinlock contention by the operating system. From the operating system's perspective, all service requests arrive sequentially and therefore obviate the operating system's spinlock contention resolution mechanisms. According to the present invention, during the period between a lock request and a lock grant for a shared resource, a respective process is allowed to perform other useful work that does not need access to the shared resource. Also during this period, the processor executing a respective process can suspend the respective process and execute another process that performs useful work that does not need the share resource. Each process requesting a lock grant is informed of the expected delay period, placed on a reservation queue, and assigned a reservation identifier. After releasing the lock, the process uses the reservation queue to locate the next pending process to receive the lock.

More specifically, embodiments of the present invention include a method in a multi-processor system having a group of processors coupled to a bus, a shared computer readable memory unit coupled to the bus, and a shared resource with an associated lock used for resource access synchronization. the method of queuing processes for access to the lock including the steps of: executing a plurality of processes by the group of processors; intercepting a first process of the plurality of processes requesting access to the lock of the shared resource; issuing to the first process a reservation identifier indicating a queue position of the first process with respect to processes stored in a reservation queue of the memory unit; storing an identifier of the first process and the reservation identifier into a entry of the reservation queue; enabling the processes in the reservation queue, including the first process, to perform useful tasks while pending in the reservation queue; and upon a second process relinquishing the lock, using the reservation queue to identify a third process in the reservation queue and issuing an operating system service request to grant the third process access to the lock.

BRIEF DESCRIPTION OF THE DRAWINGS

FIG. 1 is a logical diagram of spinlocked processes in a queue of a prior art operating system implemented arbitration system.

FIG. 2A is a logical block diagram of a general purpose multi-processor system environment operable for the present invention arbitration procedure.

FIG. 2B is an exemplary multi-processor system using a symmetric multi-processor (SMP) architecture.

FIG. 2C is a high level block diagram of processes and a spin lock queue of the present invention within the application space of a computer system and also illustrates the operating system space.

FIG. 3 illustrates elements stored in computer readable memory used by the arbitration procedure of the present invention.

FIG. 4 is a flow diagram of steps performed by the present invention when a process of the multi-processing system originates a lock call and enters the enter gate procedure.

FIG. 5 is a flow diagram of steps performed by the present invention when a process enters the exit gate procedure.

FIG. 6A and FIG. 6B represent a flow diagram of steps of the shared resource lock request arbitration procedure of the present invention.

FIG. 7 is a flow diagram of steps performed by the present invention when a process releases a lock.

FIG. 8 is a logical diagram of queued processes that are allowed to perform useful work within the arbitration procedure of the present invention.

DETAILED DESCRIPTION OF THE INVENTION

In the following detailed description of the present invention, a computer controlled system and method for performing access arbitration between multiple processes of a multi-processor system whereby queued processes are allowed to perform useful work, numerous specific details are set forth in order to provide a thorough understanding of the present invention. However, it will be obvious to one skilled in the art that the present invention may be practiced without these specific details. In other instances well known methods, procedures, components, and circuits have not been described in detail as not to unnecessarily obscure aspects of the present invention.

Notation and Nomenclature

Some portions of the detailed descriptions which follow are presented in terms of procedures, logic blocks, processing, and other symbolic representations of operations on data bits within a computer memory. These descriptions and representations are the means used by those skilled in the data processing arts to most effectively convey the substance of their work to others skilled in the art. A procedure, logic block, process, step, etc., is here, and generally, conceived to be a self-consistent sequence of steps or instructions leading to a desired result. The steps are those

requiring physical manipulations of physical quantities. Usually, though not necessarily, these quantities take the form of electrical or magnetic signals capable of being stored, transferred, combined, compared, and otherwise manipulated in a computer system. It has proven convenient at times, principally for reasons of common usage, to refer to these signals as bits, values, elements, symbols, characters, terms, numbers, or the like.

It should be borne in mind, however, that all of these and similar terms are to be associated with the appropriate physical quantities and are merely convenient labels applied to these quantities. Unless specifically stated otherwise as apparent from the following discussions, it is appreciated that throughout the present invention, discussions utilizing terms such as "processing" or "computing" or "calculating" or "determining" or "displaying" or the like, refer to the action and processes of a computer system (e.g., **112** of FIG. 2A), or similar electronic computing device, that manipulates and transforms data represented as physical (electronic) quantities within the computer system's registers and memories into other data similarly represented as physical quantities within the computer system memories or registers or other such information storage, transmission or display devices.

System Environment

Refer to FIG. 2A and FIG. 2B which illustrate components of a multi-processor computer system **112**. Within the following discussions of the present invention, certain processes (e.g., processes **400**, **500**, **600**, and **700**) and steps are discussed that are realized, in one embodiment, as a series of instructions (e.g., software program) that reside within computer readable memory units of system **112** and executed by processors of system **112**. When executed, the instructions cause the computer system **112** to perform specific actions and exhibit specific behavior which is described in detail to follow.

In general, computer system **112** of FIG. 2A used by the present invention comprises an address/data bus **100** for communicating information, multiple processors in processor group **101** which are coupled with the bus **100** for processing information and instructions, a shared computer readable volatile memory unit **102** (e.g., random access memory, static RAM, dynamic, RAM, etc.) coupled with the bus **100** for storing information and instructions for the processors **101**, a shared computer readable non-volatile memory unit (e.g., read only memory, programmable ROM, flash memory, EPROM, EEPROM, etc.) coupled with the bus **100** for storing static information and instructions for the processor(s) **101**. System **112** also includes a shared mass storage computer readable data storage device **104** (hard drive or floppy) such as a magnetic or optical disk and disk drive coupled with the bus **100** for storing information and instructions. Optionally, system **112** can include a display device **105** coupled to the bus **100** for displaying information to the computer user, an alphanumeric input device **106** including alphanumeric and function keys coupled to the bus **100** for communicating information and command selections to the processors **101**, a cursor control device **107** coupled to the bus for communicating user input information and command selections to the processors **101**, and a signal generating device **108** coupled to the bus **100** for communicating command selections to the processors **101**.

FIG. 2B illustrates a logical block diagram **110** of the multi processors **101** in more detail in a symmetric multiprocessor (SMP) configuration in conjunction with certain shared resources. It is appreciated that the SMP configuration is exemplary only and that the lock arbitration procedure of the present invention is well suited for application to a variety of multi-processor environments. In the SMP configuration **110**, eight processors **101(1)–101(8)** are shown. A portion of their shared (RAM) memory **102***a* is also shown. Shared memory **102***a* contains data structures that are shared between the processors **101(1)–101(8)**. An input/output (I/O) shared resource **104***a*, e.g., a disk drive, is also shown. A third exemplary shared resource **109** is also shown. Any of the shared resources **102***a*, **104***a*, and **109** can be accessed by any processor of processors **101(1)–101(8)** after its associated access request is synchronized by obtaining a lock. It is appreciated that in the general case, access to the lock is performed for synchronization purposes only and is not thereafter required for further use of the shared resource by the process. Therefore, the period of time that a particular process holds the associated lock is a predetermined and readily available value.

Within the SMP configuration **110** of FIG. 2B, each processor of processors **101(1)–101(8)** is capable of simultaneously executing a respective process. A scheduler within the operating system, which is one of the executed processes, determines which processor will execute which process and when in accordance with well know techniques. In order to avoid the spinlock contention between processors **101(1)–101(8)**, the present invention implements the arbitration mechanism that is described below.

Lock Arbitration Mechanism of the Present Invention

It is appreciated that the arbitration mechanism **52** of the present invention sits on top of the operating system **60** and is application specific. As such, this arbitration mechanism **52** resides on the application side **55** of system **112** and is not part of the operating system code. In this way, processors control their own process while pending for a shared resource lock and are not placed into spinlock contention. The arbitration mechanism **52**, as further described below, calls the operating system **60** at determined points when a lock is known to be available for a given process.

Specifically, FIG. 2C illustrates a high level block diagram of the environment of the present invention. Processors **101(1)–101(8)** can be executing multiple processes, e.g., processes 1–n (referenced **41≡45**), simultaneously where n is equal to or less than the number of processors within **101(1)–101(8)**. Process arbitration mechanisms **52** of the present invention (as well as the processes 1–n themselves) reside within the application space **55** of system **112**. The other space of system **112** being the operating system space **60**.

Rather than allowing each process 1–n to call the operating system **60** directly to obtain a lock, the present invention contains process arbitration mechanisms **52** which intercept the calls via interfaces **41***a*–**45***a* and maintain a queue **220** situated within the application side **55**. Each calling process of 1–n that does not immediately get a lock granted, is informed (via interfaces **41***a*–**45**) of its place in the queue **220** situated within **52**. During the time while a process of 1–n is waiting in the queue **220** of the present invention, it can perform useful processing tasks. The arbitration mechanisms **52** of the present invention then interface directly with the operating system **60** and request locks from the operating system **60** via interface **60***a* using well known lock requesting and granting mechanisms. Since queuing is performed by **52**, the operating system **60** is only

requested for a lock when one is available. By performing process queuing in the application side **55**, the present invention advantageously eliminates spinlock contention created by the operating system **60** within processes **1**–n and therefore within processors **101(1)**–**101(8)**.

FIG. **3** illustrates components within the lock arbitration mechanism of the present invention that are maintained within a memory portion **102***b* of memory unit **102** (FIG. **2**A) and accessible by all processors **101(1)**–**101(8)**. The present invention maintains a reservation queue **220** stored in memory ("a memory queue") for each lock of a shared resource. For exemplary purposes, the following discussion describes arbitration with respect to a data structure lock that is required to perform I/O operations. However, it is appreciated that the present invention is equally well suited for application to any lock of any shared resource within a multi-processor environment **112**.

The reservation queue **220** of FIG. **3** contains a number of entries **220(0)**–**220**(n) that each relate to a respective queued process and each store two items: (1) a process identifier for the queued process, PROC_ID, stored in column **224**; and (2) a reservation identifier, RES_ID, assigned to the queued process by the present invention and stored in column **226**. Each time a process requests a lock call, the name or "identifier" of the process is stored in column **224** of a vacant entry of reservation queue **220** and a reservation identifier is obtained from a latest pending service counter **210** and also stored in the vacant entry in column **226**. The latest pending service counter **210** is then incremented. Therefore, reservation queue **220** contains a listing of all processes and their reservation identifiers that have indicated that they require access to the lock but have not yet been granted the lock.

The latest pending service counter (with a latest pending service register) **210** contains the value of the latest reservation identifier assigned to a process stored in the memory queue **220**. The reservation identifiers are sequential and indicate the place in line assigned to a particular process within reservation queue **220**. The reservation ordering maintained within the reservation queue **220** is a first come, first serviced mechanism, e.g., also known as first in, first out (FIFO).

FIG. **3** also contains an in-service counter and register **215**. The in-service counter value **215** contains the reservation identifier for the process that currently has access to the lock. Arrow **230** indicates that the in-service counter **215** is updated based on entries within reservation queue **220** leaving to access the lock. Arrow **240** indicates that processes with the latest pending service count are added to the vacant entries of reservation queue **220** and are pending access to the lock. A process can determine the relative length of the reservation queue, e.g., its position in line, by comparing the latest pending service count it receives from the latest pending service counter **210** to the count value in the in-service counter **215**.

FIG. **4** illustrates a flow diagram of steps of process **400** performed within the present invention for placing a process of system **112** into the enter gate routine **600** to handle a lock call service request made by the process. Process **400** is implemented as application program instructions stored in computer readable memory units of system **112** and executed by a processor of system **112**. At step **410**, the present invention monitors all call requests in system **112** to determine if any process executed within processors **101(1)**–**101(8)** requests access to a shared resource lock. If not, step **410** loops back to check again periodically. One exemplary way in which lock calls are detected at step **410** is by searching for $ENQ locking service requests ($ENQ lock calls) which are issued by processes operating within the DEC OpenVMS operating system **60**. If a process is detected at step **410** requesting a shared resource lock, then at step **420**, the normal operating system lock arbitration mechanism is halted and process **600** is executed (FIG. **6**A and FIG. **6**B). Process **400** then returns to step **410**. The enter gate process **600** is described in more detail below and involves the placement of the calling process into the reservation queue **220** provided the shared resource lock is busy. In this way, the shared resource lock arbitration mechanism of the present invention supplants the operating system's built in arbitration.

FIG. **5** illustrates a flow diagram of steps performed within the present invention for placing a process into the exit gate routine **700** which handles the case when a process is finished with a shared resource lock and is ready to release it to another process stored within the reservation queue **220**. Process **500** is implemented as application program instructions stored in computer readable memory units of system **112** and executed by a processor of system **112**. At step **510**, process **500** commences when a process first receives its requested shared resource lock. At step **515**, the present invention checks if the respective process is finished with the shared resource lock and ready to release the lock. If not, step **515** loops until the process is finished with the lock at which time exit gate routine **700** (FIG. **7**) is entered. Process **500** then returns.

FIG. **6**A and FIG. **6**B illustrate steps within the enter gate routine **600** of the present invention. Routine **600** is implemented as application program instructions stored in computer readable memory units of system **112** and executed by a processor of system **112**. Routine **600** is an application-specific process and operates apart from the operation system of system **112**. Process **600** avoids operating system spinlock contention by ordering processes' operating service requests using the reservation queue **220**. In accordance with the present invention, an operating system service request for a shared resource lock is not executed by a process until the lock is in effect ready to be granted to the process by the operating system **60**.

Process **600** is invoked in response to a process requesting a shared resource lock as detected within FIG. **4**. Throughout the discussion of process **600**, this invoking process is called the current process. At step **610** of FIG. **6**A, the count in the latest pending service counter **210** (FIG. **3**) is incremented by one in response to the current process requesting the lock. Also at step **610**, the present invention receives a process identifier (or name) of the current process. At step **615**, the present invention then issues a reservation identifier to the current process. This reservation identifier is the current count in the latest pending service counter **210** (after the increment of step **610**). At step **620**, the present invention identifies a vacant entry within reservation queue **220** and in that entry stores: (1) the process identifier of the current process; and (2) the reservation identifier assigned to the current process at step **620**. At the completion of step **620**, the current process is assigned a place in line within reservation queue **220**. The place in line is determined by the reservation identifier assigned to the current process.

At step **625**, the present invention checks if the shared resource lock requested by the current process is already granted to another process (gate busy) or if it is available. If the lock is already granted, then step **625** functions as a lock gate, and step **640** (FIG. **6**B) is entered. At step **625**, if the shared resource lock is available for use, then step **630** of

FIG. 6A is entered. At step **630**, the present invention stores the reservation identifier of the current process into the in-service counter to represent that the current process is granted the lock.

At step **635** of FIG. 6A, process **600** returns by issuing a standard shared resource lock call to the operating system **60** and the operating system **60** then grants the shared resource lock to the current process. At step **635**, an operating system call for the shared resource lock is only made with the advance knowledge that the lock is in fact free to be granted to the current process by the operating system **60**. In this way, operating system spinlock contention is completely avoided as the processor executing the current process is never placed into spinlock by the operating system **60**.

At step **640** of FIG. 6B, the present invention has determined that the shared resource lock requested by the current process is not available. In these cases, the current process must wait for the lock to become available, but can do useful work while waiting in accordance with the present invention. At step **640**, the present invention determines an estimated or expected waiting time until the current process will be granted the lock. This expected waiting time is based on the difference between the reservation identifier of the current process and the current count in the in-service counter **215**. This difference, M, yields the number of processes that are pending within the reservation queue **220** ahead of the current process for this shared resource lock.

As discussed above, the shared resource lock is needed by a process only during a short synchronization phase, which synchronizes access to the resource, and is not thereafter needed while the process then accesses the shared resource. For instance, a shared resource lock to an I/O data structure (in memory) **102***a* is required only to reserve a portion of memory for the process and is thereafter not needed by the process when filling the portion of memory with data during the bulk of the I/O operation. Synchronization is required so that two or more processes do not request the same memory portion at the same time. Therefore, the time each process needs the lock is a constant and known period of time and does not depend on the amount of time the process requires to perform its I/O operation (which can very widely). If the period of time that a certain process needs a particular lock is expressed as T, and the number of processes ahead of the current process is M, then the expected waiting time as determined at step **640** can be expressed as:

$$\text{Expected_Waiting_Time} \; M*T.$$

Step **640** of FIG. 6B, after computing the Expected_Waiting_Time, returns this value to the current process.

Depending on the length of the Expected_Waiting_Time, the current process can perform one of three different actions to adaptively maximize the processor cycles available within system **112**.

First, as indicated by step **645**, if the Expected_Waiting_Time is of a relatively short predetermined duration (e.g., the current process is next in line), then the present invention will delay for some small predetermined amount of time at step **660**. This small predetermined amount of time is adjusted such that it is relatively close, but shorter, than the expected waiting time. Subsequently, after step **660**, step **680** is entered whereupon the current process sits idle for a brief period until an asynchronous interrupt places the current process into step **685**. The activation of the synchronous interrupt is described with reference to FIG. **7** and originates from another process releasing the shared resource lock. Of the three different actions possible for the current process, this first action is the only action that does not allow the current process, or its processor, to perform any useful work while waiting for the lock grant. This is the case because the expected waiting time is generally set to an amount too short to perform any useful work.

In one embodiment, the small predetermined amount of time is that amount of time a process requires to hold the lock before releasing it. In another embodiment, the short duration can be adaptively adjusted such that it is set just smaller than the smallest time required to perform any useful task within a task queue (described further below). At step **685**, the process **600** returns by issuing an operating system service lock call to grant the current process the lock.

The second possible action taken by the current process is determined at step **650**, where if the Expected_Waiting_Time is in excess of a predetermined long duration (e.g., the current process **40** or more processes from the next in line) then step **670** is entered. At step **670**, the current process enters a period of timed sleep that corresponds to the duration just under the Expected_Waiting_Time. During step **670**, the processor that is executing the current process suspends the current process. This processor is then allowed to execute other processes that do not need this shared resource and can perform other useful work involving time-bounded compute-only tasks that do not require operating system services, such as formatting data packets, calculating checksums, and moving data in memory **102**. These tasks can originate from the task queue (described below). It is appreciated that at the completion of the timed sleep period of step **670**, the present invention then re-enters step **640** and is thereafter generally placed into step **660**. Therefore, at step **670**, the present invention allows processors associated with processes within the memory queue **220** to perform useful work by executing other processes rather than being forced into spinlock modes. This increases overall system efficiency.

The third possible action taken by the current process is determined at step **650**. At step **650**, if the Expected_Waiting_Time is not too long, then it is of an intermediate time period and step **655** is entered. At step **655**, the present invention allows the current process to execute other useful tasks that can be completed given the duration of the Expected_Waiting_Time. These other useful tasks can involve time-bounded compute-only tasks that do not require operating system services, such as formatting data packets, calculating checksums, and moving data in memory **102**. Therefore, at step **655**, the present invention allows processes within the reservation queue **220** to perform useful work rather than being forced into spinlock modes. This, like step **670**, increases overall system efficiency.

In one embodiment, the present invention maintains a global list of generic work items that can be performed without operating system service requests. These generic work items or tasks are maintained in the task queue discussed above. The task queue maintains a listing of task names and the expected processing time required for each to complete. As processes become available to perform useful work at step **655**, they can index this task queue and select and execute as many tasks as possible given their Expected_Waiting_Time. Furthermore, a processor within step **670** can also access this task queue to select tasks to execute.

At the completion of step **655**, the current process enters step **680** where it is momentarily forced into an idle state until it receives an asynchronous interrupt which forces entry to step **685**. This asynchronous interrupt originates

from the exit gate routine **700** of FIG. **7** and informs the current process that it is first in line to receive the lock. At this point, processing flows to step **685** of FIG. **6B** where the current process is granted the lock. As described below, the process originating the asynchronous interrupt updates the in-service counter before step **685** is invoked. In some cases, a process within step **655** can be interrupted by the asynchronous interrupt. In this case, step **685** is entered.

In an alternative embodiment, in lieu of an asynchronous interrupt, all processes within the reservation queue **220** that are held in step **680** compare their reservation identifiers to the current value within the in-service counter **215**. When the in-service counter **215** matches a particular process' reservation identifier, that process then obtains the lock via step **685**.

It is appreciated that process **600** is shown with respect to a current process, however, multiple processes are handled by process **600**. In effect, all processes stored in the reservation queue **220** can exist simultaneously within a state governed by a step of steps **660**, **670**, **655** or **680**. Therefore, all processes within the reservation queue **220** can be within process step **655** performing useful work while waiting for a lock grant.

FIG. **7** illustrates steps of the exit gate routine **700** of the present invention. Routine **700** is implemented as application program instructions stored in computer readable memory units of system **112** and executed by a processor of system **112**. Routine **700** is followed by any process that is relinquishing a shared resource lock. At step **710**, the present invention receives the reservation identifier of the process that is returning the lock. At step **715**, the present invention then compares this reservation identifier with all entries in column **226** of the reservation queue **220** to locate the entry within the reservation queue **220** corresponding to this returned reservation identifier. This located entry of the reservation queue **220** is then made vacant. At step **720**, the in-service counter **215** is then incremented by one.

At step **725** of FIG. **7**, the present invention then compares all entries in column **226** of the reservation queue **220** with the value of the in-service counter **215**. The entry matching the in-service counter **215** is then determined to be the process in the front of the line and next to receive the lock grant. At step **725**, the present invention determines the process identifier from this matching entry. At step **730**, the present invention generates an asynchronous interrupt to the process having the process identifier as determined from step **725**. This gives the process an indication that it can go ahead and the take the lock via an operating system call. The exit gate routine **700** then exits. In the alternative embodiment discussed with respect to FIG. **6B** that does not use the asynchronous interrupt, step **730** is not required since each process in the reservation queue **220** is self-monitoring for the lock release.

An advantage of the shared resource lock arbitration mechanism of the present invention is that no modifications need be made on the operation system's lock call procedures. The present invention ensures that when the operation system's lock call procedures are executed, no contention will exist between processes for the share resource lock.

FIG. **8** illustrates an exemplary time snap shot of several processes pending in the reservation queue **220** for the shared resource lock. Queue positions **810(1)**–**810(n)** correspond to processes stored within respective entries of reservation queue entries **220(1)** to **220(n)**. Gate queue **810** represents the process possessing lock. In contrast to the queue **10** of FIG. **1**, since processors **110(1)**-**110(8)** are not placed into spinlock pending their lock grant, many more processes can be placed in the queue **220** in accordance with the present invention. In this case, processes **1** . . . n are queued where n can be very much larger than 8.

More importantly, while pending, the processors **110(1)** –**110(8)** are allowed to perform useful work within system **112**. For instance, process **10** executing over CPU **4** (e.g., processor **110(4)**), is shown in queue position **810(2)**. While pending for its lock grant, process **10** is free to perform useful work or processor **110(4)** is free to execute another process that can perform useful work. Similarly, process **9** executing over CPU **3** (e.g., processor **110(3)**), is shown in queue position **810(4)**. While pending for its lock grant, process **9** is free to perform useful work or processor **110(3)** is free to execute another process that can perform useful work. The same is true with respect to the other queue positions of **810(1)**–**810(n)** as shown.

One embodiment of the present invention has been implemented on a DEC OpenVMS using its $ENQ locking services as an example. This locking service is used very frequently by the TPC-C benchmark application (in excess of 25K operations per second) and most other business application programs. Without the present invention, an 8-processor Turbolaser computer system wastes an equivalent of 1.5 processors in spinlock mode for the lock service spinlock. With present invention shared resource lock arbitration mechanism, this waste is reduced to almost zero. The overhead of the present invention is one enter and exit call pair that jacket each of the common $ENQ and the $DEQ service calls. This overhead translates to a waste of less than one quarter of one processor.

The preferred embodiment of the present invention, a lock arbitration mechanism for a multi-processor system, is thus described. While the present invention has been described in particular embodiments, it should be appreciated that the present invention should not be construed as limited by such embodiments, but rather construed according to the below claims.

What is claimed is:

1. In a multi-processor system having a group of processors coupled to a bus, a shared computer readable memory unit coupled to said bus, and a shared resource with an associated lock used for resource access synchronization, a method of queuing processes for access to said lock comprising the steps of:

a) executing a plurality of processes by said group of processors;

b) intercepting a first process of said plurality of processes requesting access to said lock of said shared resource;

c) issuing to said first process a reservation identifier indicating a queue position of said first process with respect to processes stored in a reservation queue of said memory unit;

d) storing an identifier of said first process and said reservation identifier into a entry of said reservation queue;

e) enabling said processes in said reservation queue, including said first process, to perform useful tasks while pending in said reservation queue; and

f) upon a second process relinquishing said lock, using said reservation queue to identify a third process in said reservation queue and issuing an operating system service request to grant said third process access to said lock.

2. A method as described in claim **1** wherein said step b) comprises the steps of:

monitoring said multi-processor system for $ENQ lock calls originating from said plurality of processes; and

intercepting a $ENQ lock call originating from said first process.

3. A method as described in claim 1 wherein said step c) comprises the steps of:

upon said first process requesting access to said lock of said shared resource, incrementing a latest pending service counter value maintained in said memory unit;

accessing said latest pending service counter value to obtain said reservation identifier; and

issuing said reservation identifier to said first process.

4. A method as described in claim 1 wherein said step e) comprises the step of enabling said first process to perform a useful compute-only task that does not require operating system services, such as: formatting data packets; calculating checksums; and moving data in said memory unit, while said first process is pending in said reservation queue.

5. A method as described in claim 4 further comprising the steps of:

determining an expected waiting time before said first process is to obtain said lock;

accessing a task queue to obtain a respective useful task that can be completed within said expected waiting time; and

said first process executing said respective useful task while said first process is pending in said reservation queue for said lock.

6. A method as described in claim 1 wherein said step e) comprises the step of placing said first process to sleep and enabling a processor executing said first process to execute a fourth process that performs a useful compute-only task that does not require operating system services, such as: formatting data packets; calculating checksums; and moving data in said memory unit, while said first process is pending in said reservation queue.

7. A method as described in claim 1 where said step f) comprises the steps of:

upon said second process relinquishing said lock, incrementing an in-service counter to indicate a next reservation identifier;

searching said reservation queue to locate said third process having said next reservation identifier; and

generating an asynchronous interrupt to said third process causing said operating system to grant said third process said lock.

8. A multi-processor system comprising a group of processors, a shared computer readable memory unit, and a shared resource with an associated lock used for resource access synchronization between a plurality of processes, said memory unit containing instructions that when executed implement a method of queuing processes for access to said lock, said method comprising the steps of:

a) executing said plurality of processes by said group of processors;

b) intercepting a first and a second process, of said plurality of processes, requesting access to said lock of said shared resource;

c) issuing to said first process a first reservation identifier that indicates a queue position of said first process with respect to processes stored in a reservation queue of said memory unit and issuing to said second process a second reservation identifier;

d) storing an identifier of said first process and said first reservation identifier into a first entry of said reservation queue and storing an identifier of said second process and said second reservation identifier into a second entry of said reservation queue;

e) enabling said first process and said second process to perform useful tasks while pending in said reservation queue; and

f) upon a third process relinquishing said lock, using said reservation queue to identify said first process in said reservation queue and issuing an operating system service request to grant said first process access to said lock.

9. A multi-processor system as described in claim 8 wherein said step b) comprises the steps of:

monitoring said multi-processor system for $ENQ lock calls from said plurality of processes;

intercepting a first $ENQ lock call originally from said first process; and

intercepting a second $ENQ lock call originally from said second process.

10. A multi-processor system as described in claim 8 wherein said step c) comprises the steps of:

upon said first process requesting access to said lock of said shared resource, incrementing a latest pending service counter value maintained in said memory unit;

accessing said latest pending service counter to obtain said first reservation identifier;

issuing said first reservation identifier to said first process;

upon said second process requesting access to said lock of said shared resource, incrementing said latest pending service counter value maintained in said memory unit;

accessing said latest pending service counter to obtain said second reservation identifier; and

issuing said second reservation identifier to said first process.

11. A multi-processor system as described in claim 8 wherein said step e) comprises the step of enabling said first process to perform a first useful compute-only task that does not require operating system services, such as: formatting data packets; calculating checksums; and moving data in said memory unit, while said first process is pending in said reservation queue.

12. A multi-processor system as described in claim 11 wherein said method further comprises the steps of:

determining an expected waiting time before said first process is to obtain said lock;

accessing a task queue to obtain said first useful task that can be completed within said expected waiting time; and

said first process executing said first useful task while said first process is pending in said reservation queue for said lock.

13. A multi-processor system as described in claim 11 wherein said step e) further comprises the step of placing said second process to sleep and enabling a processor executing said second process to execute a fourth process that performs a second useful compute-only task that does not require operating system services, such as: formatting data packets; calculating checksums; and moving data in said memory unit, while said second process is pending in said reservation queue.

14. A multi-processor system as described in claim 8 where said step f) comprises the steps of:

upon said third process relinquishing said lock, incrementing an in-service counter value to indicate a next reservation identifier;

searching said reservation queue to locate said first process having said next reservation identifier; and

generating an asynchronous interrupt to said first process causing said operating system to grant said first process said lock.

15. A shared memory unit in a multi-processor system having group of processors coupled to a bus and a shared resource with an associated lock used for resource access synchronization, said shared memory unit containing instructions stored therein that, when executed, cause said system to implement a method of queuing processes to access said lock comprising the steps of:

a) executing a plurality of processes by said group of processors;

b) intercepting a first process of said plurality of processes requesting access to said lock of said shared resource;

c) issuing to said first process a reservation identifier indicating a queue position of said first process with respect to processes stored in a reservation queue of said memory unit;

d) storing an identifier of said first process and said reservation identifier into a entry of said reservation queue;

e) enabling said processes in said reservation queue, including said first process, to perform useful tasks while pending in said reservation queue; and

f) upon a second process relinquishing said lock, using said reservation queue to identify a third process in said reservation queue and issuing an operating system service request to grant said third process access to said lock.

16. A shared memory unit as described in claim 15 wherein said step b) comprises the steps of:

monitoring said multi-processor system for $ENQ lock calls originating from said plurality of processes; and

intercepting a $ENQ lock call originating from said first process.

17. A shared memory unit as described in claim 15 wherein said step c) comprises the steps of:

upon said first process requesting access to said lock of said shared resource, incrementing a latest pending service counter value maintained in said memory unit;

accessing said latest pending service counter value to obtain said reservation identifier; and

issuing said reservation identifier to said first process.

18. A shared memory unit as described in claim 15 wherein said step e) comprises the step of enabling said first process to perform a useful compute-only task that does not require operating system services, such as: formatting data packets; calculating checksums; and moving data in said memory unit, while said first process is pending in said reservation queue.

19. A shared memory unit as described in claim 18 wherein said method further comprising the steps of:

determining an expected waiting time before said first process is to obtain said lock;

accessing a task queue to obtain a respective useful task that can be completed within said expected waiting time; and

said first process executing said respective useful task while said first process is pending in said reservation queue for said lock.

20. A shared memory unit as described in claim 15 wherein said step e) comprises the step of placing said first process to sleep and enabling a processor executing said first process to execute a fourth process that performs a useful compute-only task that does not require operating system services, such as: formatting data packets; calculating checksums; and moving data in said memory unit, while said first process is pending in said reservation queue.

21. A shared memory unit as described in claim 15 where said step f) comprises the steps of:

upon said second process relinquishing said lock, incrementing an in-service counter to indicate a next reservation identifier;

searching said reservation queue to locate said third process having said next reservation identifier; and

generating an asynchronous interrupt to said third process causing said operating system to grant said third process said lock.

* * * * *

US005835762A

United States Patent [19]

Gans et al.

[11] Patent Number: 5,835,762

[45] Date of Patent: *Nov. 10, 1998

[54] **METHOD AND APPARATUS FOR PROCESSING ELECTRONIC MAIL IN PARALLEL**

[75] Inventors: **Steven S. Gans**, San Francisco; **Farzad Nazem**, Redwood City, both of Calif.

[73] Assignee: **Oracle Corporation**, Redwood Shores, Calif.

[*] Notice: The term of this patent shall not extend beyond the expiration date of Pat. No. 5,504,897.

[21] Appl. No.: **939,998**

[22] Filed: **Sep. 29, 1997**

Related U.S. Application Data

[63] Continuation of Ser. No. 660,737, Jun. 6, 1996, abandoned, which is a continuation of Ser. No. 465,734, Jun. 6, 1995, abandoned, which is a continuation of Ser. No. 175,159, Dec. 22, 1994, Pat. No. 5,504,897.

[51] **Int. Cl.**[6] **G06F 9/40**

[52] **U.S. Cl.** **395/670**

[58] **Field of Search** 395/670–678, 395/680

[56] **References Cited**

U.S. PATENT DOCUMENTS

4,084,228	4/1978	Dufond et al.	395/673
4,796,178	1/1989	Jennings et al.	395/673
4,805,134	2/1989	Calo et al.	707/10
4,896,290	1/1990	Rhodes et al.	395/887
5,093,918	3/1992	Heyen et al.	395/200.45
5,138,653	8/1992	Le Clerq	379/93.24
5,177,680	1/1993	Tsukino et al.	704/1
5,212,793	5/1993	Donica et al.	395/675
5,245,532	9/1993	Moutier	364/400
5,278,984	1/1994	Batchelor	395/200.37
5,283,856	2/1994	Gross et al.	706/47
5,325,310	6/1994	Johnson et al.	395/200.36
5,333,266	7/1994	Boaz et al.	395/200.36
5,394,549	2/1995	Stringfellow et al.	395/670
5,404,501	4/1995	Carr et al.	395/680
5,406,557	4/1995	Baudoin	370/407
5,487,100	1/1996	Kane	455/31.2

OTHER PUBLICATIONS

Dr. Thomas, Rebecca, "Writing to the Net, Killing Processes by Name", Unix World, p. 137; vol. VIII, No. 11, Nov. 1991.

Rochkind, Marc J., "Advanced Unix Programming", Prentice–Hall, pp. 6, 7, 219, 220, 1985.

Meyer, Steve, "Tuning Your NetWare 3.11 Server: Key parameters for internal process management", Lan Times Jan. 24, 1994; pp. 25; vol. 11, Issue 2.

Stevens, Richard W., "Advanced Programming in the UNIX Environment", Addison–Wesley, 1992, 1992, pp. 188, 189.

Reiss, Levi, *Unix System Administration Guide*, Osborne McGraw Hill 1993, pp. 424, 425, 82, 91.

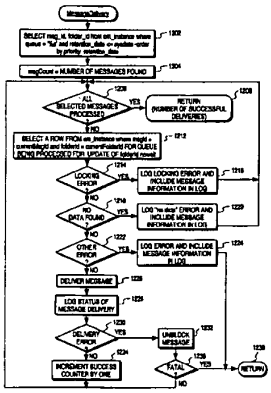
Reinhardt, Andy, *Smarter E–Mail is Coming*, Mar. 1993, Byte Magazine Cover Story, p. 90.

Primary Examiner—Alvin E. Oberley
Assistant Examiner—St. John Courtenay III
Attorney, Agent, or Firm—Blakely, Sokoloff, Taylor & Zafman LLP

[57] **ABSTRACT**

The present invention provides a method and apparatus for processing electronic mail in parallel. The present invention provides the ability to process mail objects in an electronic mail system in parallel. A message can be assigned to a queue. One or more processes can manage a plurality of messages in the queue. Each process can identify the next entry to be processed. Entries previously processed can be marked such that subsequent access is locked out. Mail objects and process information can be stored in a relational database system that provides the ability to perform locking at the record level. A process can be configured to perform a plurality of activities in a plurality of time periods. A guardian process can initiate or terminate other processes based on process information. Further, a guardian process examines system information periodically and identifies any need to initiate, restart, or stop one or more processes. Further, the guardian process can pass process information to an initiated process. One or more tables can be used to retain message information such as a instance table that includes a queue column. Further, tables can be used to retain process information such as process, process parameters, and process time tables.

20 Claims, 24 Drawing Sheets



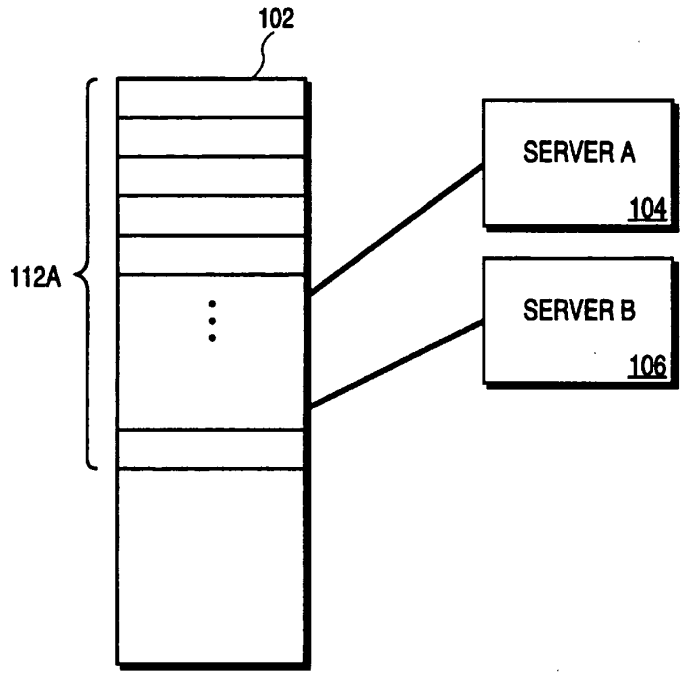


FIG. 1A

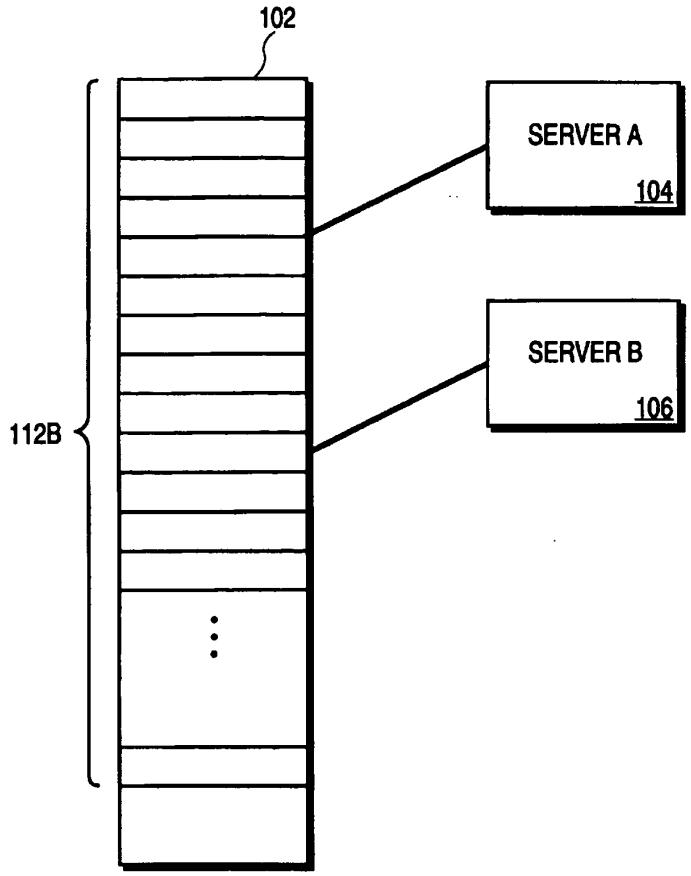


FIG. 1B

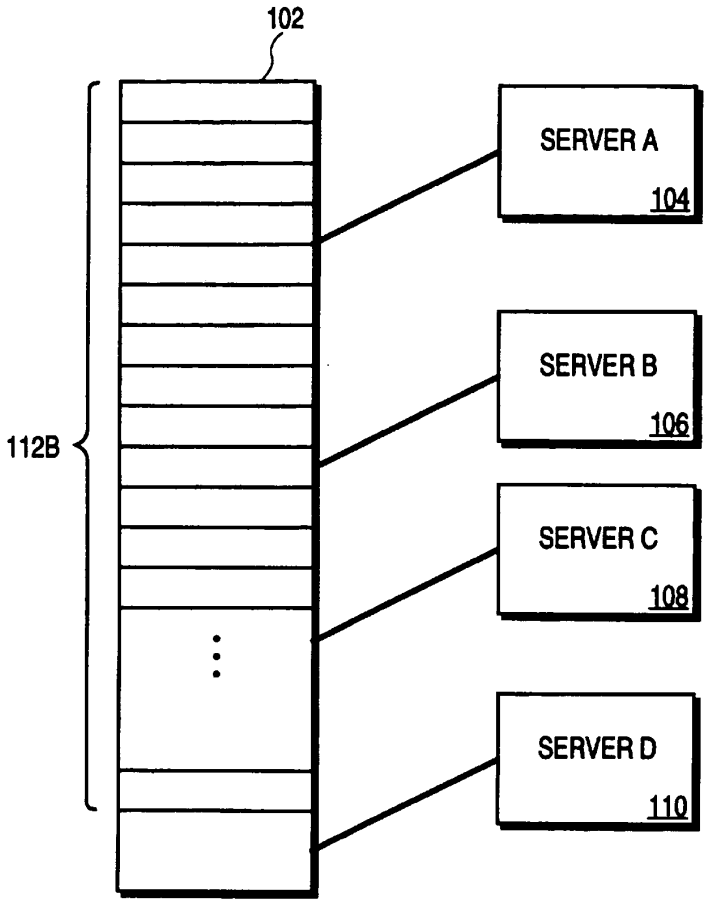


FIG. 1C

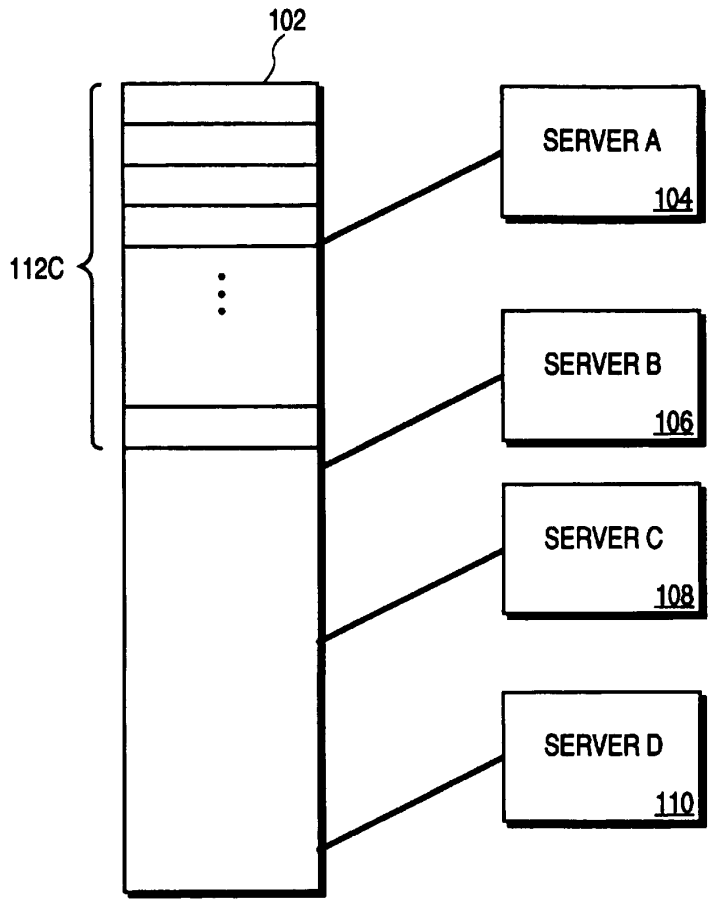


FIG. 1D

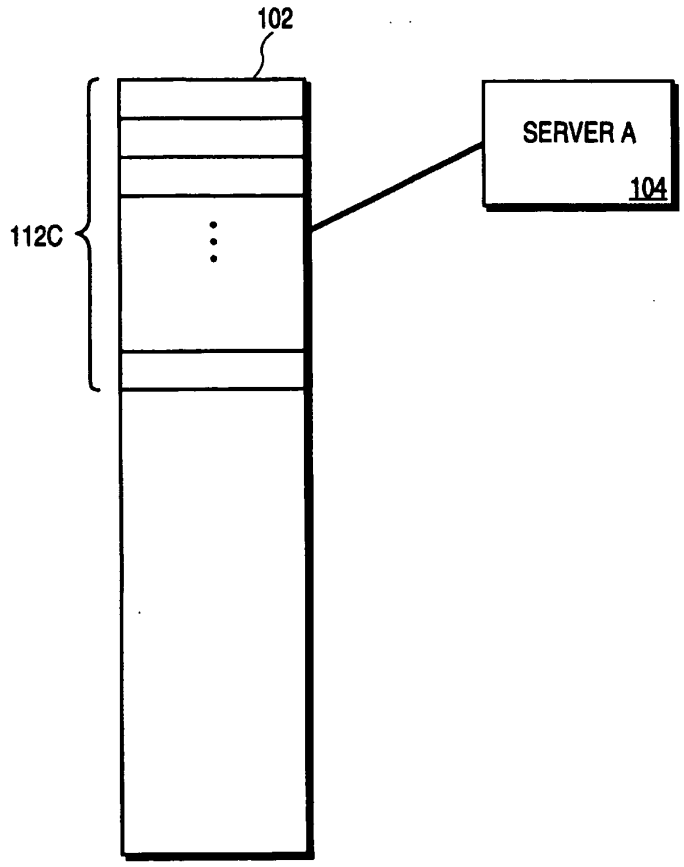


FIG. 1E

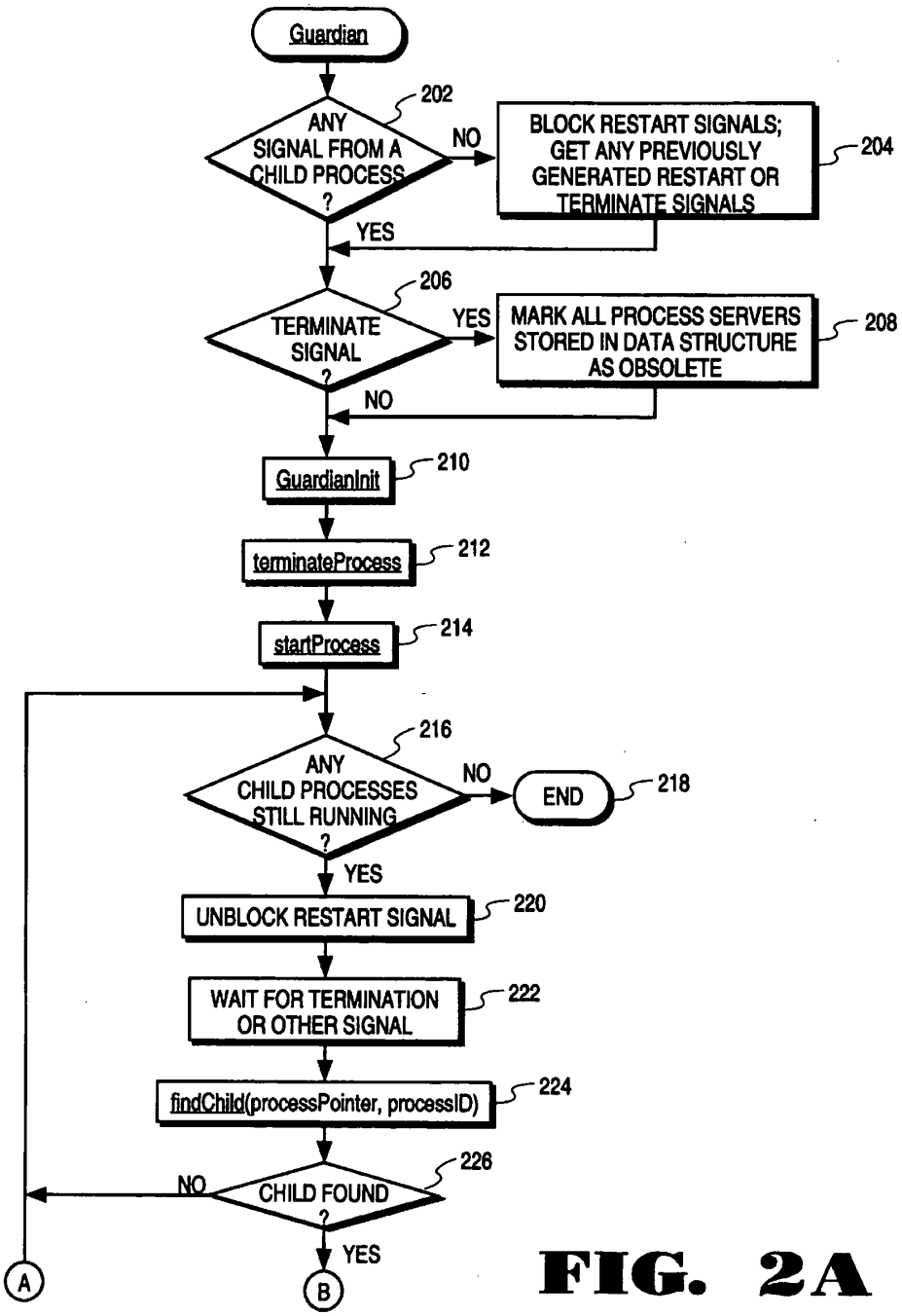

Guardian
202
ANY SIGNAL FROM A CHILD PROCESS ?
NO
BLOCK RESTART SIGNALS; GET ANY PREVIOUSLY GENERATED RESTART OR TERMINATE SIGNALS
204
YES
206
TERMINATE SIGNAL ?
YES
MARK ALL PROCESS SERVERS STORED IN DATA STRUCTURE AS OBSOLETE
208
NO
GuardianInit
210
terminateProcess
212
startProcess
214
216
ANY CHILD PROCESSES STILL RUNNING ?
NO
END
218
YES
UNBLOCK RESTART SIGNAL
220
WAIT FOR TERMINATION OR OTHER SIGNAL
222
findChild(processPointer, processID)
224
226
NO
CHILD FOUND ?
YES
A
B

FIG. 2A

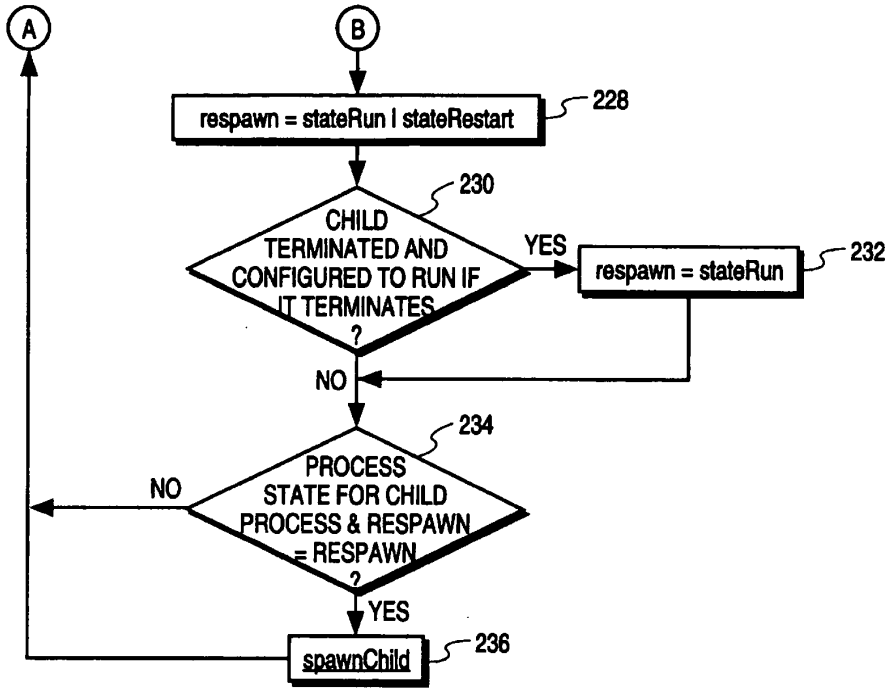

A
B
respawn = stateRun | stateRestart
228
230
CHILD TERMINATED AND CONFIGURED TO RUN IF IT TERMINATES ?
YES
respawn = stateRun
232
NO
234
PROCESS STATE FOR CHILD PROCESS & RESPAWN = RESPAWN ?
NO
YES
spawnChild
236

FIG. 2B

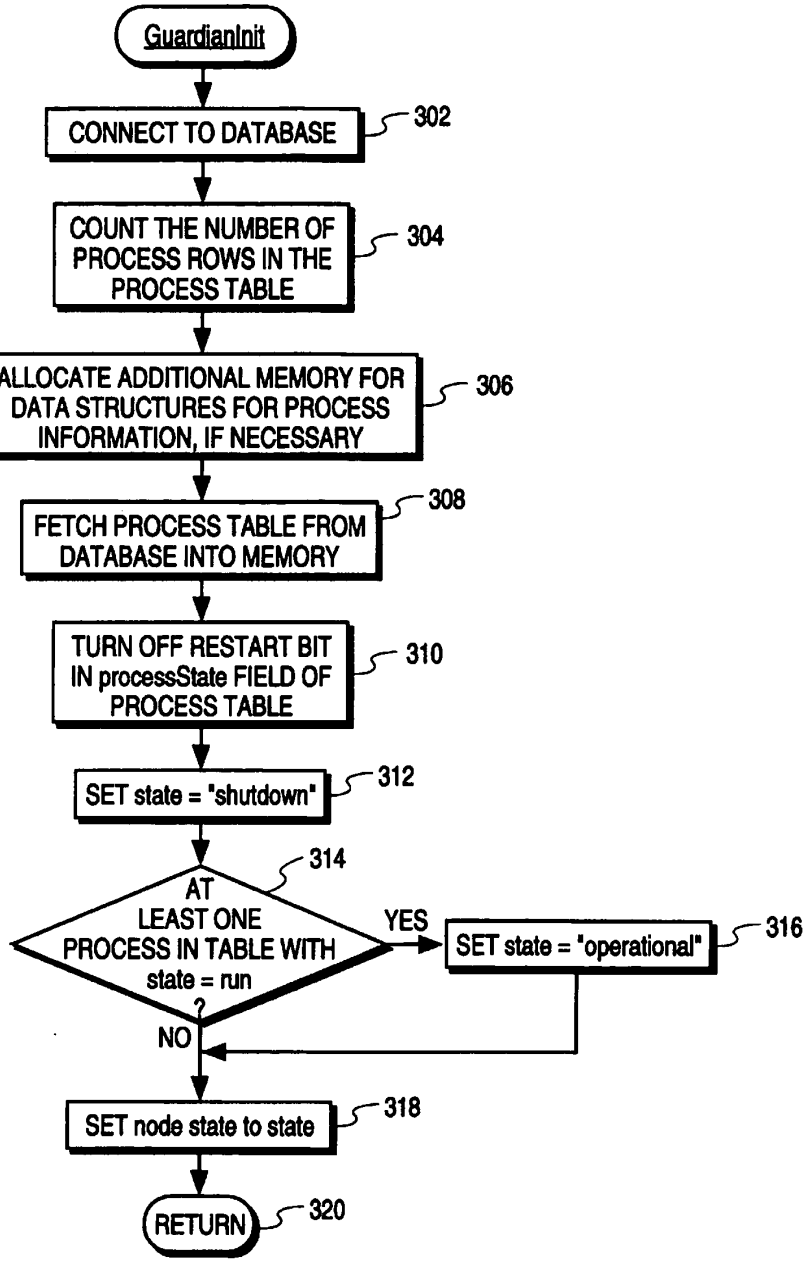

GuardianInit
CONNECT TO DATABASE
302
COUNT THE NUMBER OF PROCESS ROWS IN THE PROCESS TABLE
304
ALLOCATE ADDITIONAL MEMORY FOR DATA STRUCTURES FOR PROCESS INFORMATION, IF NECESSARY
306
FETCH PROCESS TABLE FROM DATABASE INTO MEMORY
308
TURN OFF RESTART BIT IN processState FIELD OF PROCESS TABLE
310
SET state = "shutdown"
312
314
AT LEAST ONE PROCESS IN TABLE WITH state = run ?
YES
SET state = "operational"
316
NO
SET node state to state
318
RETURN
320

FIG. 3

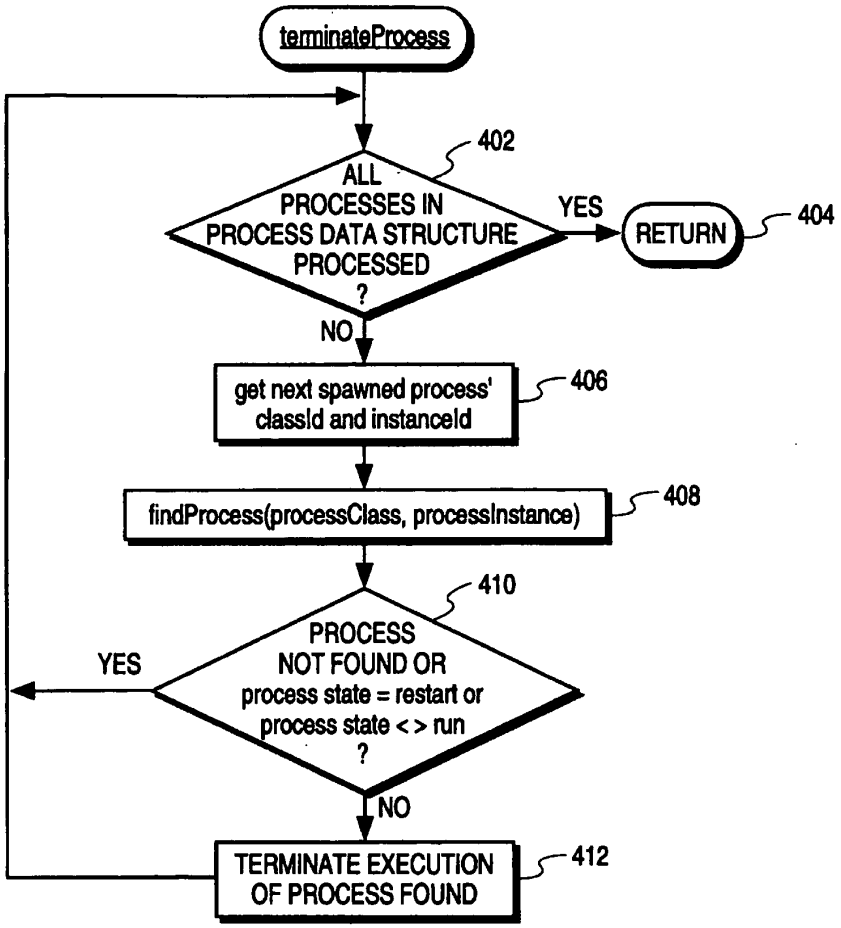

terminateProcess
402
ALL PROCESSES IN PROCESS DATA STRUCTURE PROCESSED ?
YES
RETURN
404
NO
get next spawned process' classId and instanceId
406
findProcess(processClass, processInstance)
408
410
PROCESS NOT FOUND OR process state = restart or process state < > run ?
YES
NO
TERMINATE EXECUTION OF PROCESS FOUND
412

FIG. 4

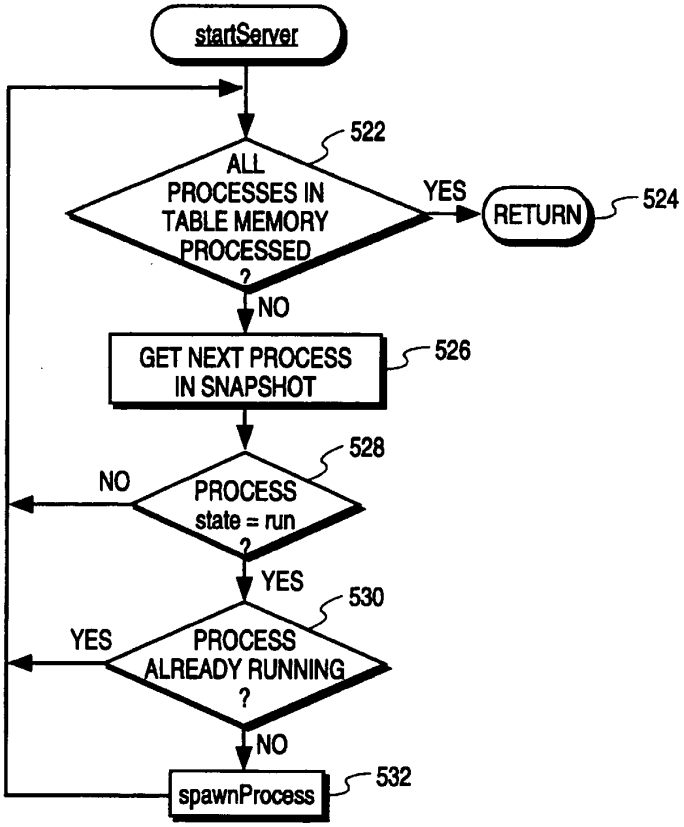

startServer
522
ALL PROCESSES IN TABLE MEMORY PROCESSED ?
YES
RETURN
524
NO
GET NEXT PROCESS IN SNAPSHOT
526
528
NO
PROCESS state = run ?
YES
530
YES
PROCESS ALREADY RUNNING ?
NO
spawnProcess
532

FIG. 5

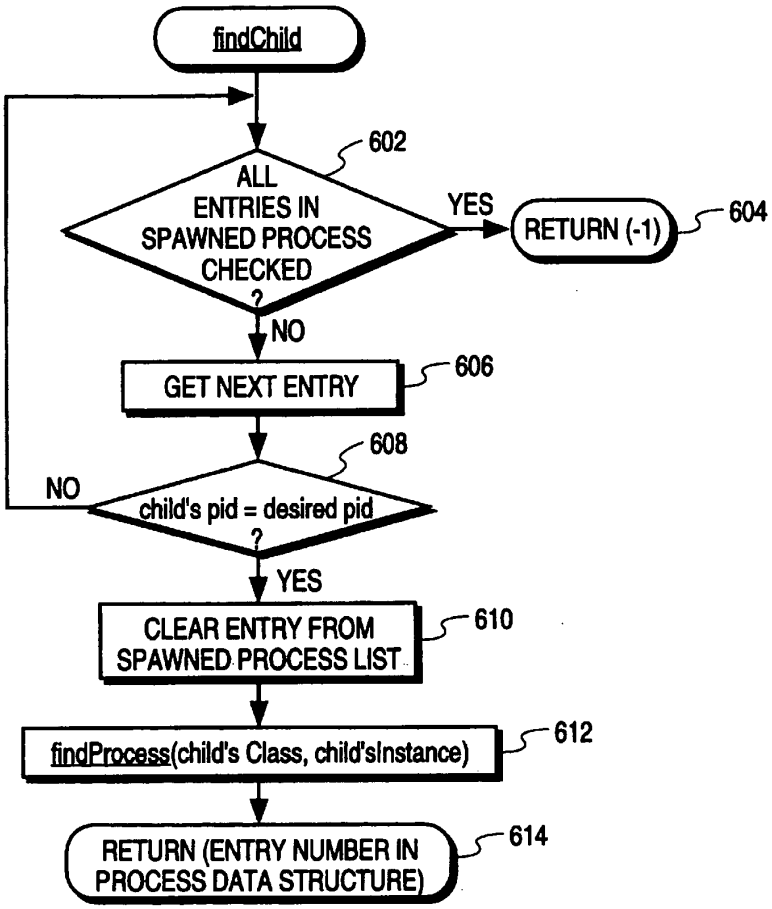

findChild
602
ALL ENTRIES IN SPAWNED PROCESS CHECKED ?
YES
RETURN (-1)
604
NO
GET NEXT ENTRY
606
608
NO
child's pid = desired pid ?
YES
CLEAR ENTRY FROM SPAWNED PROCESS LIST
610
findProcess(child's Class, child'sInstance)
612
RETURN (ENTRY NUMBER IN PROCESS DATA STRUCTURE)
614

FIG. 6

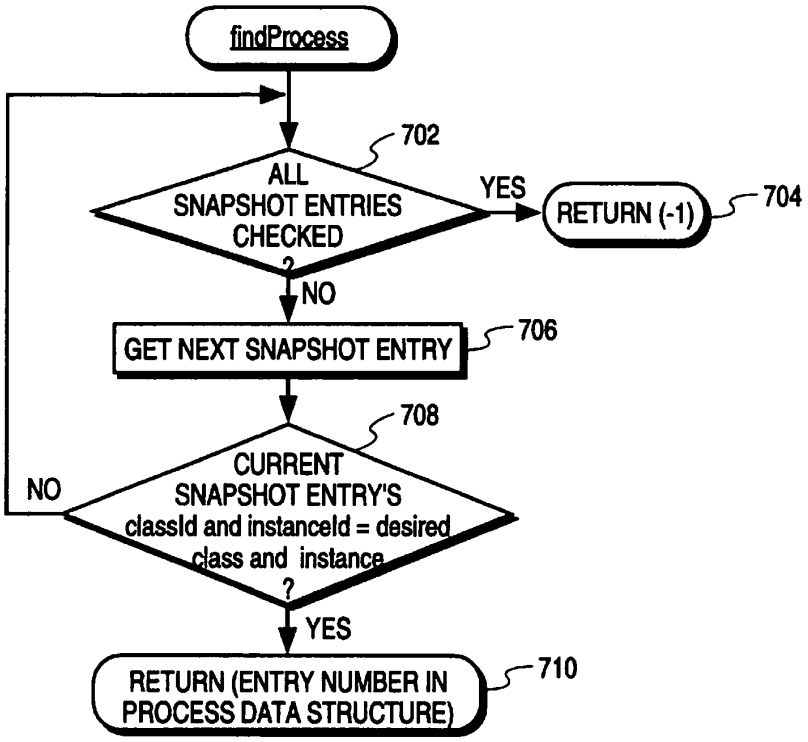

findProcess
702
ALL SNAPSHOT ENTRIES CHECKED ?
YES
RETURN (-1)
704
NO
GET NEXT SNAPSHOT ENTRY
706
708
CURRENT SNAPSHOT ENTRY'S classId and instanceId = desired class and instance ?
NO
YES
RETURN (ENTRY NUMBER IN PROCESS DATA STRUCTURE)
710

FIG. 7

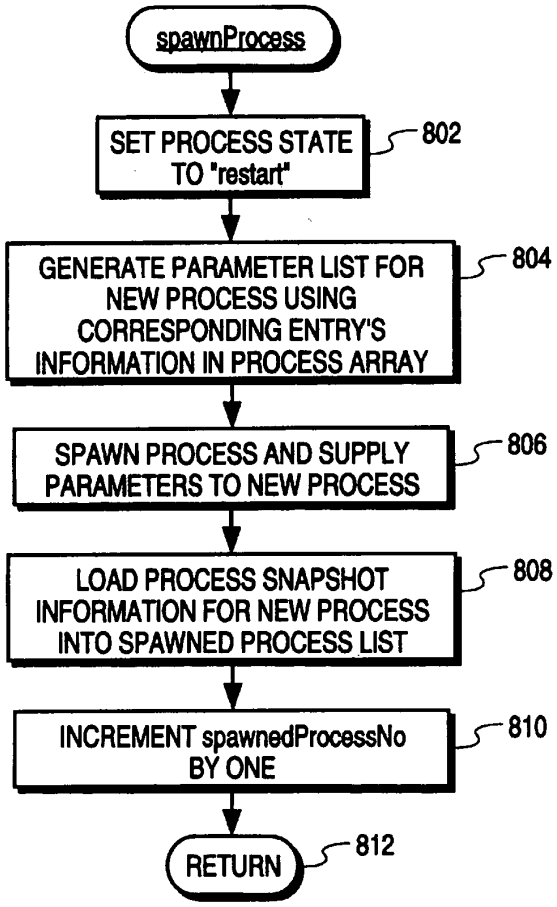

spawnProcess
SET PROCESS STATE TO "restart"
802
GENERATE PARAMETER LIST FOR NEW PROCESS USING CORRESPONDING ENTRY'S INFORMATION IN PROCESS ARRAY
804
SPAWN PROCESS AND SUPPLY PARAMETERS TO NEW PROCESS
806
LOAD PROCESS SNAPSHOT INFORMATION FOR NEW PROCESS INTO SPAWNED PROCESS LIST
808
INCREMENT spawnedProcessNo BY ONE
810
RETURN
812

FIG. 8

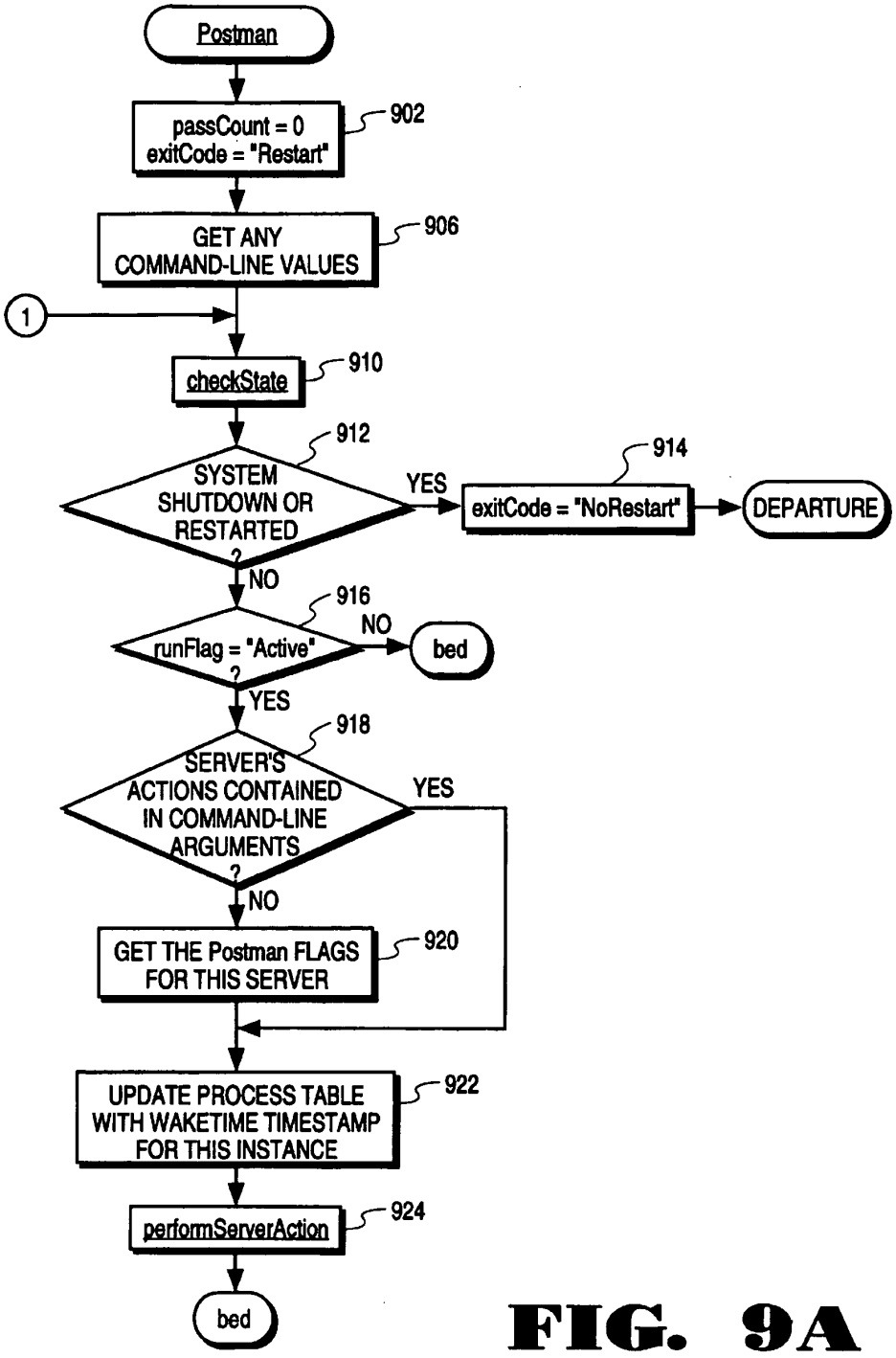

Postman
passCount = 0
exitCode = "Restart"
902
GET ANY
COMMAND-LINE VALUES
906
1
checkState
910
912
SYSTEM
SHUTDOWN OR
RESTARTED
?
YES
914
exitCode = "NoRestart"
DEPARTURE
NO
916
runFlag = "Active"
?
NO
bed
YES
918
SERVER'S
ACTIONS CONTAINED
IN COMMAND-LINE
ARGUMENTS
?
YES
NO
GET THE Postman FLAGS
FOR THIS SERVER
920
UPDATE PROCESS TABLE
WITH WAKETIME TIMESTAMP
FOR THIS INSTANCE
922
performServerAction
924
bed

FIG. 9A

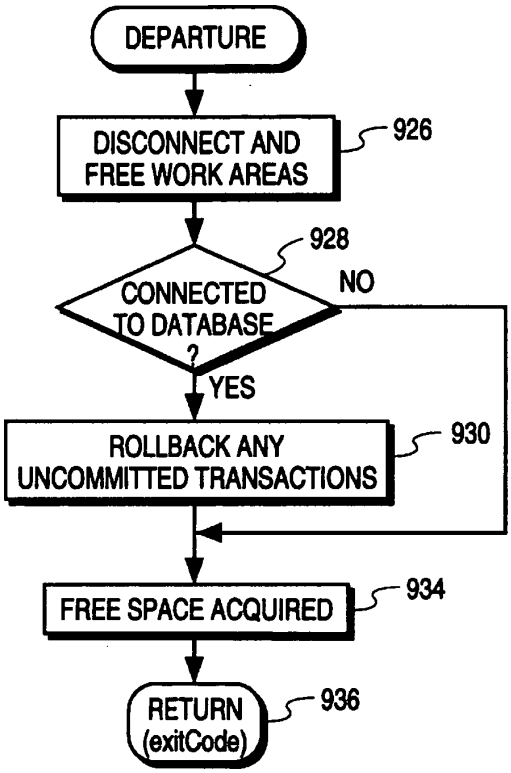

DEPARTURE
DISCONNECT AND FREE WORK AREAS
926
928
CONNECTED TO DATABASE ?
NO
YES
ROLLBACK ANY UNCOMMITTED TRANSACTIONS
930
FREE SPACE ACQUIRED
934
RETURN (exitCode)
936

FIG. 9B

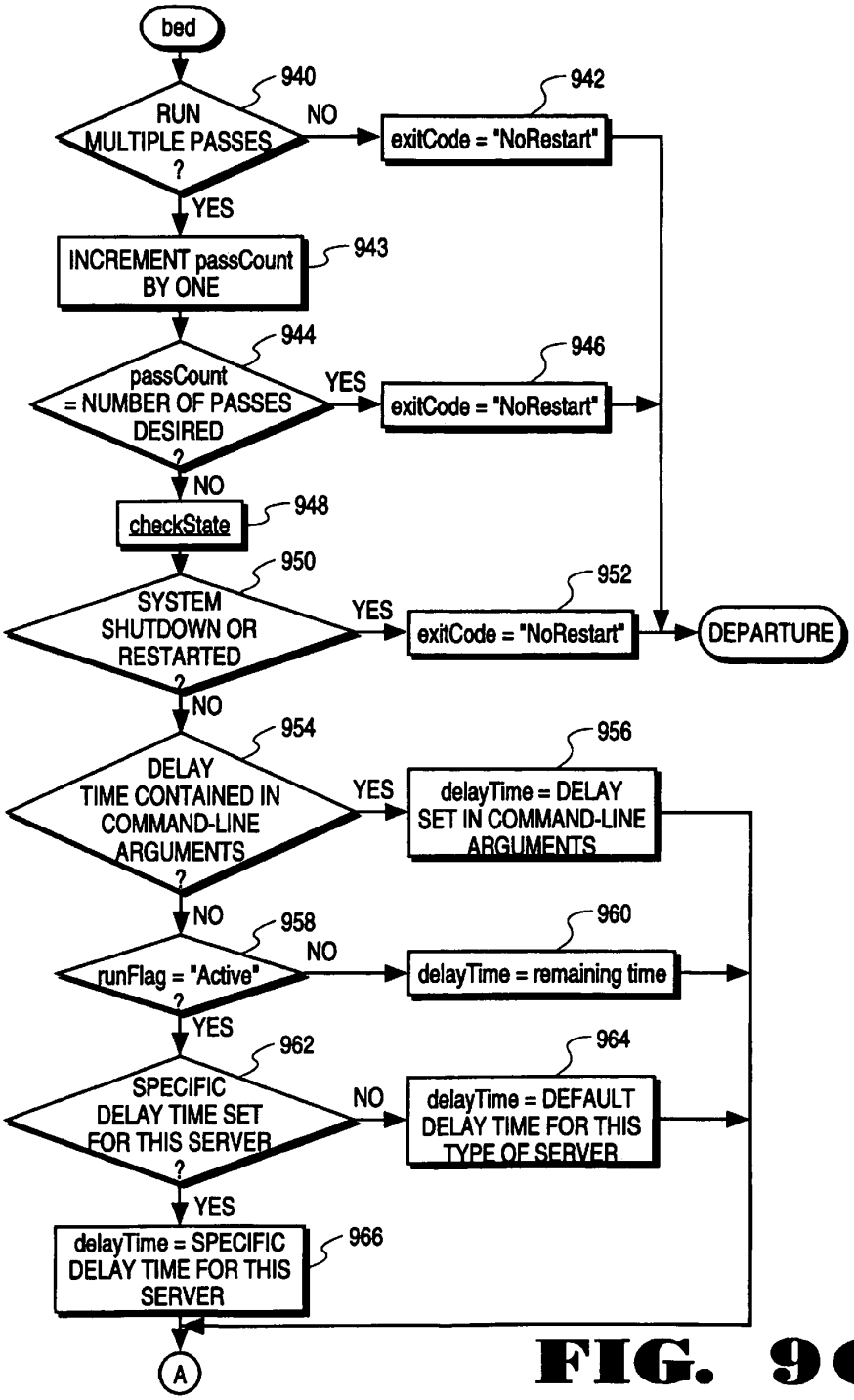

bed
940
RUN MULTIPLE PASSES ?
NO
942
exitCode = "NoRestart"
YES
INCREMENT passCount BY ONE
943
944
passCount = NUMBER OF PASSES DESIRED ?
YES
946
exitCode = "NoRestart"
NO
checkState
948
950
SYSTEM SHUTDOWN OR RESTARTED ?
YES
952
exitCode = "NoRestart"
DEPARTURE
NO
954
DELAY TIME CONTAINED IN COMMAND-LINE ARGUMENTS ?
YES
956
delayTime = DELAY SET IN COMMAND-LINE ARGUMENTS
NO
958
runFlag = "Active" ?
NO
960
delayTime = remaining time
YES
962
SPECIFIC DELAY TIME SET FOR THIS SERVER ?
NO
964
delayTime = DEFAULT DELAY TIME FOR THIS TYPE OF SERVER
YES
delayTime = SPECIFIC DELAY TIME FOR THIS SERVER
966
A

FIG. 9C

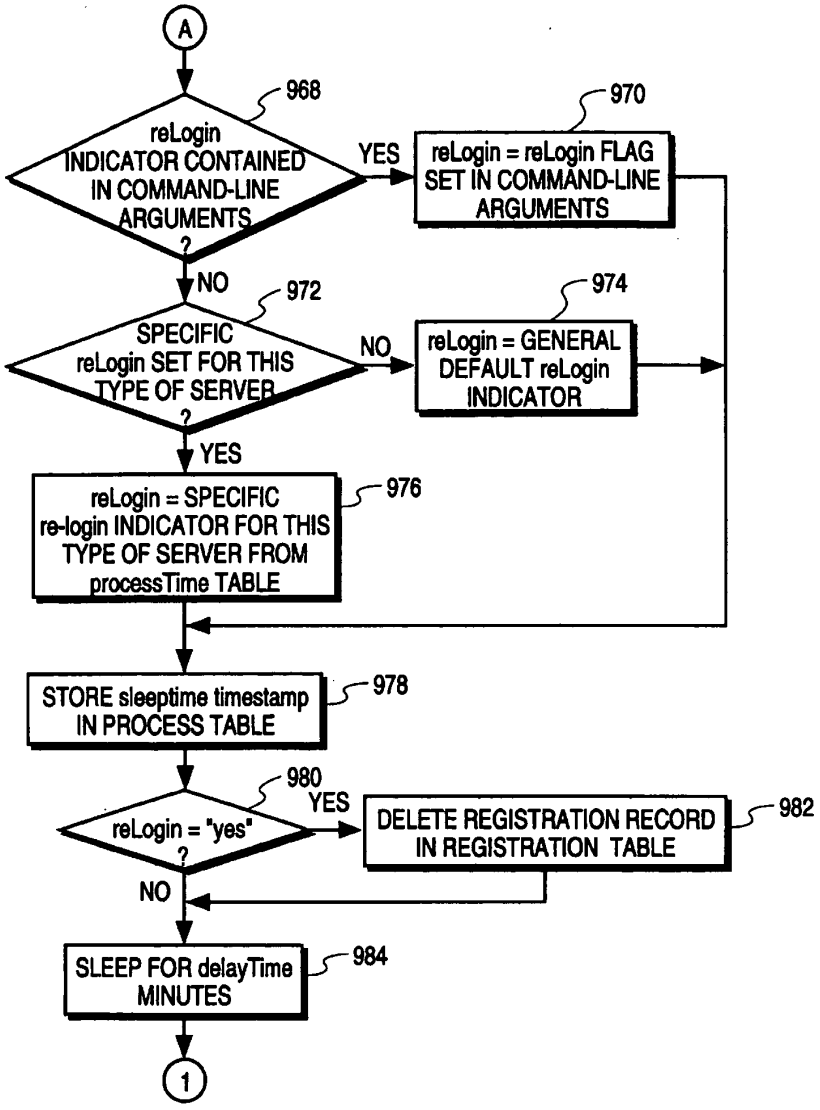

A
968
reLogin INDICATOR CONTAINED IN COMMAND-LINE ARGUMENTS ?
YES
970
reLogin = reLogin FLAG SET IN COMMAND-LINE ARGUMENTS
NO
972
SPECIFIC reLogin SET FOR THIS TYPE OF SERVER ?
NO
974
reLogin = GENERAL DEFAULT reLogin INDICATOR
YES
reLogin = SPECIFIC re-login INDICATOR FOR THIS TYPE OF SERVER FROM processTime TABLE
976
STORE sleeptime timestamp IN PROCESS TABLE
978
980
reLogin = "yes" ?
YES
DELETE REGISTRATION RECORD IN REGISTRATION TABLE
982
NO
SLEEP FOR delayTime MINUTES
984
1

FIG. 9D

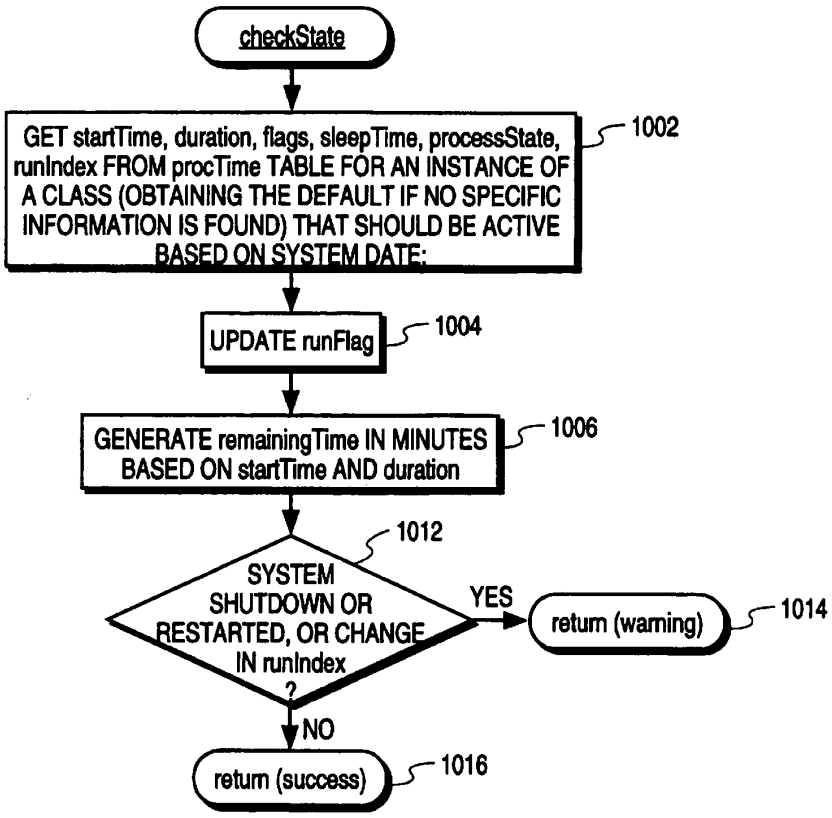

checkState
GET startTime, duration, flags, sleepTime, processState, runIndex FROM procTime TABLE FOR AN INSTANCE OF A CLASS (OBTAINING THE DEFAULT IF NO SPECIFIC INFORMATION IS FOUND) THAT SHOULD BE ACTIVE BASED ON SYSTEM DATE;
1002
UPDATE runFlag
1004
GENERATE remainingTime IN MINUTES BASED ON startTime AND duration
1006
1012
SYSTEM SHUTDOWN OR RESTARTED, OR CHANGE IN runIndex ?
YES
return (warning)
1014
NO
return (success)
1016

FIG. 10

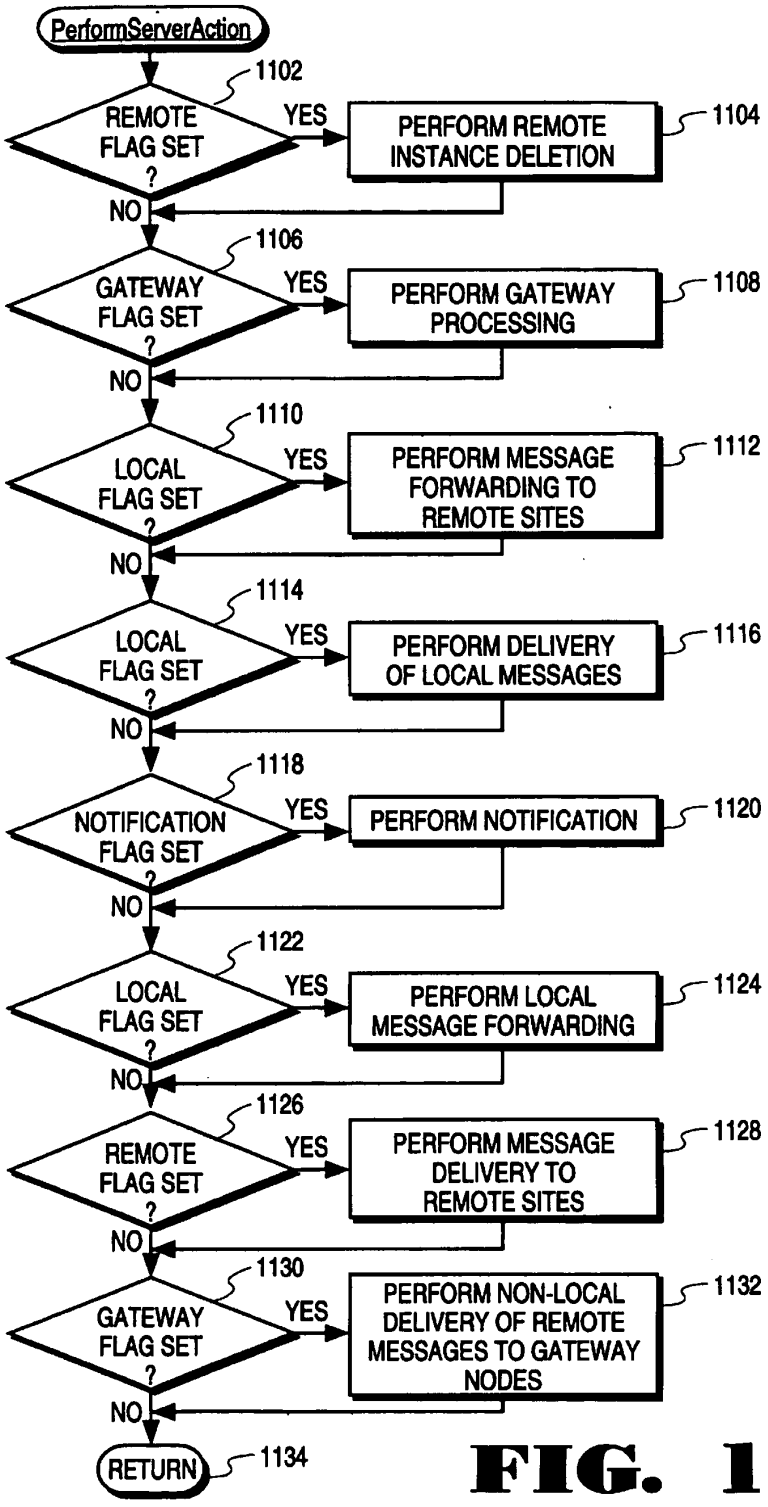
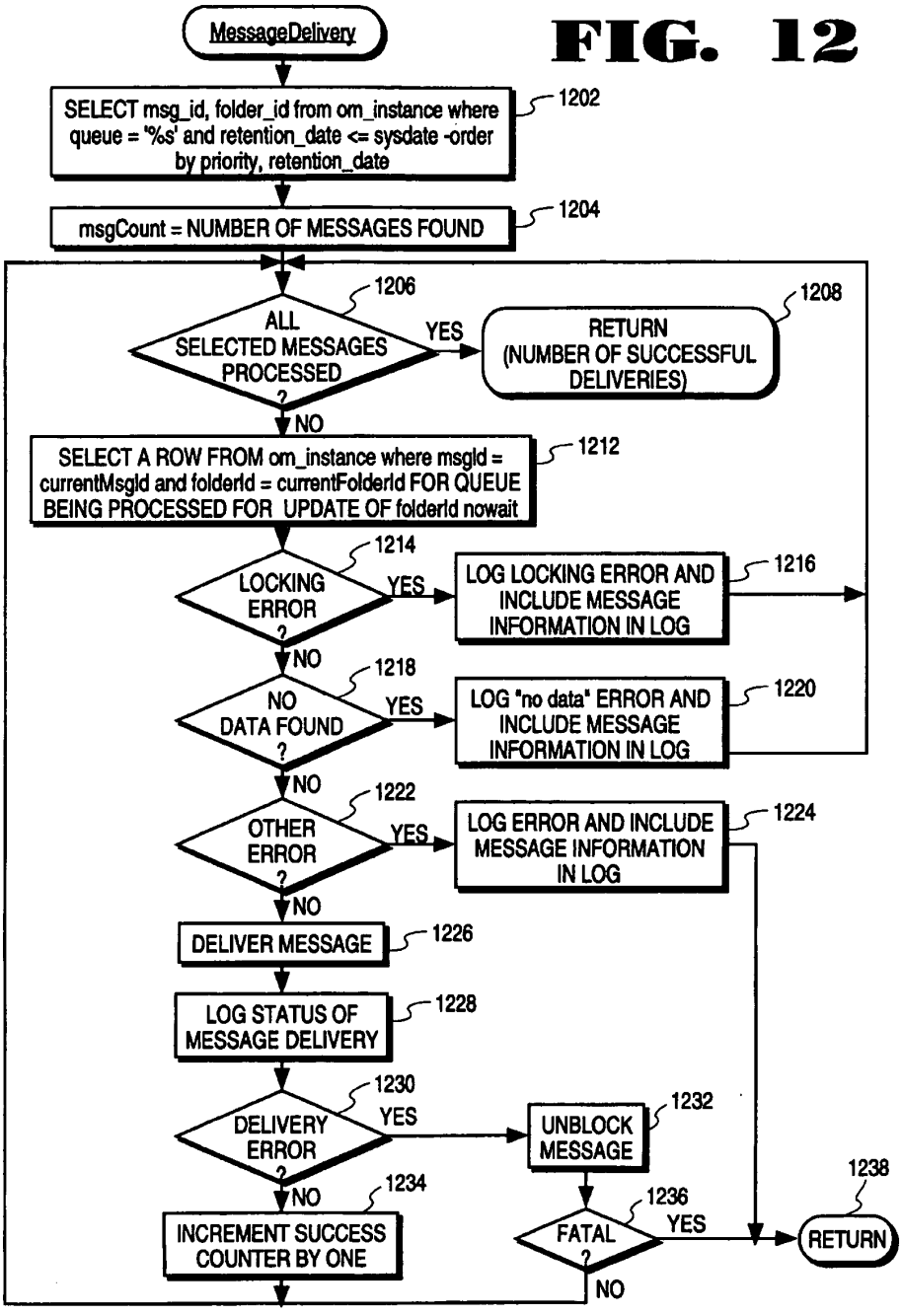

PerformServerAction
1102
REMOTE FLAG SET ?
YES
PERFORM REMOTE INSTANCE DELETION
1104
NO
1106
GATEWAY FLAG SET ?
YES
PERFORM GATEWAY PROCESSING
1108
NO
1110
LOCAL FLAG SET ?
YES
PERFORM MESSAGE FORWARDING TO REMOTE SITES
1112
NO
1114
LOCAL FLAG SET ?
YES
PERFORM DELIVERY OF LOCAL MESSAGES
1116
NO
1118
NOTIFICATION FLAG SET ?
YES
PERFORM NOTIFICATION
1120
NO
1122
LOCAL FLAG SET ?
YES
PERFORM LOCAL MESSAGE FORWARDING
1124
NO
1126
REMOTE FLAG SET ?
YES
PERFORM MESSAGE DELIVERY TO REMOTE SITES
1128
NO
1130
GATEWAY FLAG SET ?
YES
PERFORM NON-LOCAL DELIVERY OF REMOTE MESSAGES TO GATEWAY NODES
1132
NO
RETURN
1134

FIG. 11

FIG. 12


MessageDelivery
SELECT msg_id, folder_id from om_instance where queue = '%s' and retention_date <= sysdate -order by priority, retention_date
1202
msgCount = NUMBER OF MESSAGES FOUND
1204
ALL SELECTED MESSAGES PROCESSED ?
1206
YES
RETURN (NUMBER OF SUCCESSFUL DELIVERIES)
1208
NO
SELECT A ROW FROM om_instance where msgId = currentMsgId and folderId = currentFolderId FOR QUEUE BEING PROCESSED FOR UPDATE OF folderId nowait
1212
LOCKING ERROR ?
1214
YES
LOG LOCKING ERROR AND INCLUDE MESSAGE INFORMATION IN LOG
1216
NO
NO DATA FOUND ?
1218
YES
LOG "no data" ERROR AND INCLUDE MESSAGE INFORMATION IN LOG
1220
NO
OTHER ERROR ?
1222
YES
LOG ERROR AND INCLUDE MESSAGE INFORMATION IN LOG
1224
NO
DELIVER MESSAGE
1226
LOG STATUS OF MESSAGE DELIVERY
1228
DELIVERY ERROR ?
1230
YES
UNBLOCK MESSAGE
1232
NO
INCREMENT SUCCESS COUNTER BY ONE
1234
FATAL ?
1236
YES
RETURN
1238
NO

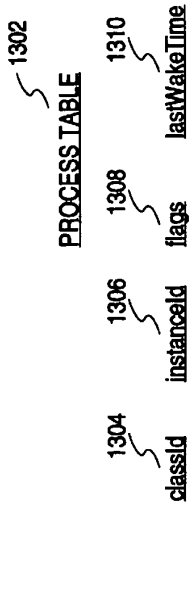


FIG. 13A

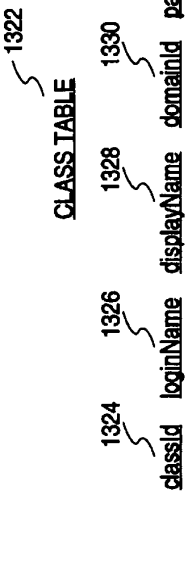


FIG. 13B

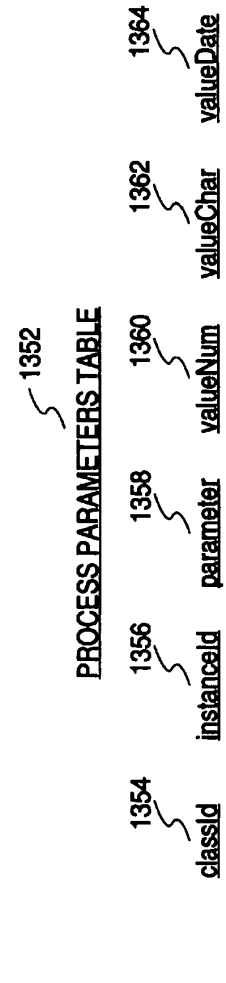

1352
PROCESS PARAMETERS TABLE
1354
classId
1356
instanceId
1358
parameter
1360
valueNum
1362
valueChar
1364
valueDate
1368A
1368N

FIG. 13C

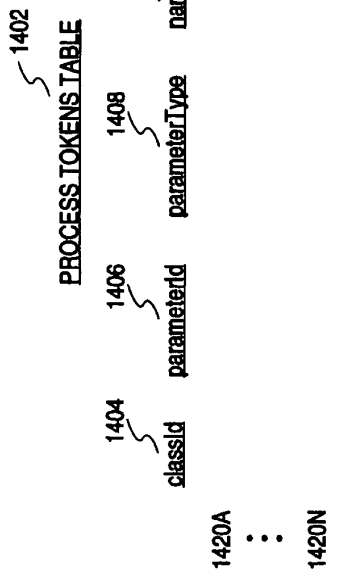


FIG. 14A

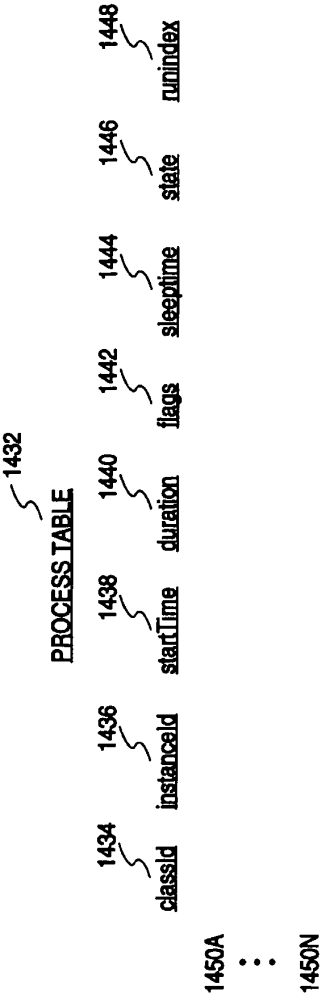


FIG. 14B

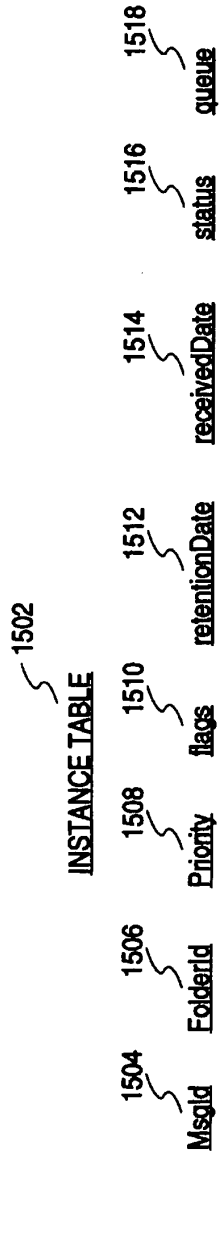

1502
INSTANCE TABLE
1504
MsgId
1506
FolderId
1508
Priority
1510
flags
1512
retentionDate
1514
receivedDate
1516
status
1518
queue
1522A
1522N

FIG. 15

METHOD AND APPARATUS FOR PROCESSING ELECTRONIC MAIL IN PARALLEL

This is a continuation of application Ser. No. 08/660,737, filed Jun. 6, 1996, now abandoned, which is a continuation of application Ser. No. 08/465,734 filed Jun. 6, 1995, now abandoned, which is a continuation of application Ser. No. 08/175,159 filed Feb. 22, 1994 Issued U.S. Pat. No. 5,504,897.

BACKGROUND OF THE INVENTION

1. Field of the Invention

This invention relates to the field of parallel processing in an electronic mail environment.

2. Background Art

Electronic mail messaging provides the ability to communicate information throughout an enterprise (e.g., send and receive messages and files between enterprise users). Electronic mail users can send, for example, mail messages, scheduling messages, directory information, and files.

Electronic mail systems provide the ability to perform mail operations. For example, electronic mail operations include the ability to send and receive messages (i.e., mail or calendar scheduling messages directory information, and/or files). Messages received by a user can be, for example, read and/or forwarded to another mail user. Further, a user can send a reply message to the sending user. Other operations may be provided to manage messages and files.

Messages in a electronic mail systems can be grouped, or queued, based on some like characteristic (e.g., the type of further processing required). For example, a submission queue can contain messages targeted for a particular location. A rerouting queue can be used to store messages that need to be routed to another location. A notification queue can contain a list of messages that have been placed in a user's incoming mail box, and for which users are to receive notification. A dead message queue can be used to identify messages that are not deliverable or returnable to the sender. A garbage collection queue can be used to contain messages that can be removed from a system. Remote queues contain messages bound for remote locations. Gateway queues contain messages destined for foreign messaging environments.

As the number of mail users increases, the number of messages to be processed by a mail system typically increases. Conversely, as the number of mail users decreases, the number of messages decreases. If, for example, messaging increases and processing capability to handle messaging remains constant, the number of messages in the message queues such as the ones discussed above can increase. Prior art systems provide the ability to serially process messages, or queue entries. However, these systems do not provide the ability to scale processing (up or down) to accommodate a change in messaging activity.

SUMMARY OF THE INVENTION

The present invention provides the ability to scale an electronic mail system. The present invention provides the ability to process mail entries in parallel to accommodate increased messaging activity. Further, the present invention provides the ability to down scale processing capability to accommodate decreases in messaging activity.

The present invention provides the ability to scale a queue such that a queue can be generic and have one or more processes manage a portion of messages in the queue.

Instead of assigning a message to a particular process, a message can be assigned to a queue. Further, multiple processes can be assigned to process a queue. Thus, as more activity causes the number of entries in a queue to increase, additional processes can be assigned to process the queue's entries. Similarly, as activity decreases and the number of queue entries decrease, the processing capability assigned to a queue can be decreased.

Each process can identify the next entry to be processed, and then process the entry. Entries previously processed can be marked such that processes that subsequently access the entry are aware that the entry has been processed. Any order for entry selection can be used. For example, queue entries can be placed in the queue in the order in which they are received. Further, priorities can be assigned to queue entries. Thus, for example, each process can select queue entries on a First In First Out (FIFO) basis. Further, the FIFO selection can be varied based on the priorities assigned to the queue entries.

FIFO

Any method can be used to identify queue entries previously or currently being processed by one process. In the preferred embodiment, messaging and process information are stored in a relational database system that provides the ability to perform locking at the record level. Such a relational database management system (RDBMS) is provided by Oracle Corporation. Messaging and process information are stored in relations, or tables, in the RDBMS.

A process can be used to perform multiple tasks or activities. Each process can be configured to perform one or more of these activities. Further, processes can be configured to run during a certain time period. Thus, for example, multiple processes can be configured to perform garbage collection. A garbage collector process can be further configured to, for example, clean up mail messages or scheduler messages, or clean up replication or directory registration information. Further, a garbage collector can be run at night to perform garbage collection on mail messages. Another garbage collector can be run during the daytime to perform garbage collection tasks.

The number and type of processes can be determined or altered by a electronic mail system administrator. The present invention can retain information related to the processes. A parent process, the guardian process, can initiate or terminate other processes. A guardian process can access process information to determine what number and type of processes to initiate. Further, the guardian process can examine the system information at an interval of time to determine what processes are running. Based on the system information and the process information, the guardian can identify any need to initiate, restart, or stop one or more processes. Further, the guardian process can pass process identification and other process information to an initiated process to assist the process in determining how to proceed.

Using a RDBMS with record locking capability, queue entries can be stored in a database with each queue entry being a row in a database relation, or table. As each entry is selected for processing, the row in the table that corresponds to the queue entry can be locked. Each process can examine a snapshot of the queue and attempt to access the next queue entry. If the entry is not locked, the entry can be selected for processing. If the entry is locked, the entry cannot be selected by a subsequent process.

One or more tables can be used to retain message information. For example, an instance table can contain an entry for each instance of a message and retain queue information. This table can be examined by the processes to identify the next message to be processed.

Additional tables can be used to retain process information. For example, a process table can contain a class designation, instance identifier, flags, timestamps (e.g., last wake time and last sleep time), and a process state (e.g., run or not run). Another table can be used to define general information for each class of processes. For example, fields in the table can be used to assign names to the executables in each class.

A process parameters table contains parameter information for a process instance or for a class of processes. A process can be configured for periods of dormancy between work cycles (i.e., performing configured tasks). A process time table is used to determine the periods in which a process is to remain dormant. For example, the table can contain information regarding the time of day that a process is to run.

BRIEF DESCRIPTION OF THE DRAWINGS

FIGS. 1A–1E illustrate mail system queues and processes.

FIG. **2** illustrates a process flow for a guardian process.

FIG. **3** illustrates a guardianInit process flow.

FIG. **4** illustrates a terminateProcess process flow.

FIG. **5** illustrates a startServer process flow.

FIG. **6** illustrates a findChild process flow.

FIG. **7** illustrates a findProcess process flow.

FIG. **8** illustrates a spawnProcess process flow.

FIGS. **9**-A–9D illustrate a postman process flow.

FIG. **10** provides an example of a checkState process flow.

FIG. **11** illustrates a performServerAction process flow for a Postman process.

FIG. **12** illustrates a local message delivery process flow including locking.

FIG. **13**A illustrates a process table.

FIG. **13**B provides an example of a class table structure.

FIG. **13**C provides an example of a process parameters table.

FIG. **14**A provides an example of a process tokens table.

FIG. **14**B provides an illustration of a process time table.

FIG. **15** provides an example of an instance queue table.

DETAILED DESCRIPTION OF THE INVENTION

A method and apparatus for processing electronic mail in parallel is described. In the following description, numerous specific details (e.g., specific table entries) are set forth in order to provide a more thorough description of the present invention. It will be apparent, however, to one skilled in the art, that the present invention may be practiced without these specific details. In other instances, well-known features have not been described in detail so as not to obscure the invention.

Electronic mail systems store mail items while they wait to be processed by the system. In the present invention, queues can be used to store mail items awaiting processing by the mail system. For example, a mail message sent by one mail user to another may be stored in multiple queues on its journey from the sender to the recipient. The message is maintained in a queue awaiting whatever processing is needed. For example, a message being sent across a gateway to a user on another mail system may be stored in a remote queue to await forwarding to the other system. Upon its arrival at the remote node, it can be placed in a rerouting queue awaiting transmittal to the appropriate queue on the remote node.

The amount of traffic in a mail system can vary. As the mail activity varies, the number of items stored in a system queue can vary. For example, when mail activity increases while the ability to process the increased mail items remains stable, the number of mail items waiting to be processed can increase. The present invention provides the ability to extend the processing capability of an electronic mail system to handle such increases in activity. That is, the present invention provides the ability to process mail entries in parallel to accommodate increased messaging activity.

Conversely, when a decrease in mail activity occurs and processing capability remains stable, some processing capability can become idle. The present invention provides the ability to scale back processing capability to accommodate the reduced mail activity.

The present invention provides the ability to scale a queue such that a queue can be generic and have one or more processes process a portion of the messages in the queue. Instead of assigning a message to a particular process, a message can be assigned to a queue. Further, multiple processes can be assigned to process a queue. FIG. **1**A illustrates a mail system queue **102** that contains mail entries **112**A. Mail entries **112**A are assigned to queue **102**. Server A **104** and Server B **106** have been configured to process mail entries in queue **102**.

Server A **104** and Server B **106** select one or more of entries **112**A in queue **102** to process. In the present invention, any selection technique can be used to select the next entry or entries to be processed by a process. For example, queue entries can be selected from the queue in the order in which they are received into the queue using a First In First Out (FIFO) method. Further, priorities can be assigned to queue entries. Thus, the selection can be made based on priorities assigned to queue entries.

Thus, using a selection technique, each process processing queue entries can identify the next entry or entries to be processed. Once an entry has been processed, it can be marked to prevent another process from processing the entry. Any method can be used to identify queue entries previously or currently being processed by one process without departing from the scope of the present invention.

In the preferred embodiment, messaging and process information are stored in a relational database system that provides the ability to perform locking at the record level. Such a relational database management system (RDBMS) is provided by Oracle Corporation. Using an RDBMS, messaging (e.g., queue entries) can be stored in relations, or tables, in the RDBMS. When a process selects a mail item (i.e., queue entry) for processing, the record that represents the item is locked. If another process attempts to select the same mail item from the queue, a locking exception is generated. Thus, subsequent processes can identify the queue entries handled by another process. Other methods for identifying items previously processed can be used without departing from the scope of the present invention.

As mail system activity increases, there is an increase in the number of mail items that must be processed by the mail system. If processing capability remains stable, the number of mail entries in a queue can increase such as is illustrated in FIG. 1B. Queue **102** now contains entries **112**B for processing by Servers A and B.

To handle the increase in queue entries, additional processes can be added as illustrated in FIG. 1C. In addition to servers A and B, servers C and D have been configured to process entries **112**B. Assuming a stable level of system activity, the additional processing capability can result in a

reduction in queue entries as illustrated in FIG. 1D. A system administrator, upon viewing the situation illustrated in FIG. 1D, can determine that some of the processing capability assigned to queue **102** is not needed and can be removed. FIG. 1E illustrates queue **102** and a reduction in the processing capability illustrated in FIG. 1D. That is, Servers B–D in FIG. 1D have been eliminated and one server (i.e., server A) remains to handle the mail entries **112**C.

Processes

Different types of processes can be used in the present invention to perform mail system tasks. The following are examples of processes and some of the tasks that can be performed in a mail system. Additional process types and tasks can be used with the present invention without departing from its scope. Examples of types of processes include: postman, scheduler, replicator, monitor, statistics, garbage collector, and guardian. A process can be used to perform multiple, or different tasks or activities. Further, processes can be configured to run during a certain time period. The number and characteristics of processes can be determined or altered by a electronic mail system administrator based on such factors as system activity levels.

A postman process, for example, delivers local mail items (e.g., scheduling and mail), remote mail items, handles triggered mail items (e.g., return receipts and auto-forward), and send notification of new messages locally. A scheduler process can be used to handle scheduling requests. A replicator process can be used to synchronize directory information. A monitor process can be used to check message flow, database space usage, and process status. A garbage collector process can remove unneeded mail items (e.g., unowned messages) and reclaim the space used for these items. A process, a guardian process, can act as the parent process for the other processes. The parent process can start and then stop the processes. It can verify that the proper number of each process type is running.

Multiple processes, for example, can be configured to perform garbage collection. A garbage collector process can be further configured to, for example, clean up mail or scheduler messages, or clean up replication or directory registration information. One of the garbage collector processes can run at night to perform garbage collection on mail messages. Another garbage collector can be run during the daytime to perform garbage collection tasks.

Data Tables

System information can be stored such that it can be referenced intermittently during processing, and at system startup. In the preferred embodiment, this information is stored in a relational database system such as the relational database management system (RDBMS) provided by Oracle Corporation. Information stored in RDBMS tables includes messaging and processing information. Specific details used to describe the type of information associated with mail and processes is only for the sake of illustration. Additional or different information can be used without departing from the scope of the invention.

—Process Information—

A process is assigned a record in a process table. This record is used by the guardian process as a request for invocation. FIG. 13A illustrates a process table. Classid **1304** contains an identification of a process class (e.g., postman). InstanceId **1306** contains a unique value within a particular process class. It differentiates among different instances of a particular class of process.

Flags field **1308** can contain any number of flags to further define a process instance. For example, flags **1308** can be used to particularize the tasks to be performed by a process instance. Thus, multiple instances of a process class can handle some subset of the total tasks defined for the class.

The flags field for an instance of the postman process, for example, can be used to indicate that the postman instance perform local delivery, remote delivery, gateway processing, or notification. To illustrate further, the flags field for an instance of a garbage collector process can be used to indicate that the process cleanup registration records, or perform scheduler, directory, or mail garbage collection.

A process instance can become dormant during execution. For example, during its active state, a process can perform its tasks. After performing its task, the process can lay dormant, or passive, for a period of time before becoming active again and performing its defined tasks. LastWakeTime **1310** is used to identify the time at which a process awoke from a dormant period. LastSleepTime **1312** is used to identify the time at which the process last entered into a dormant period. ProcessState **1314** indicates the state of a process (e.g., whether or not the process should be run).

Class information is stored in the class table. The guardian process can, for example, use the information contained in this table to determine names for an executable in each class. FIG. 13B provides an example of a class table structure. ClassId **1324** has the same meaning as in the process table. LoginName **1326** and password **1332** are used to authenticate the login to, for example, the RDBMS.

DisplayName **1328** is used to identify a process class, for example, in a configuration or management panel or report. DomainId **1330** can be used for gateways (i.e., a link between systems with different protocols) and for user-defined applications as defined by ApplicationId field **1334**. ExecName **1336** identifies the name of an executable module (i.e., a module capable of execution in the system). Subsystem field **1338** can be used to group together a variety of individual processes into a single module (i.e., mail or schedular).

A guardian process is responsible for invoking a process and passing to the initiated process its process class and instanceId value. A guardian process can access process information to determine what number and type of processes to initiate.

Further, the guardian process can examine the system information at an interval of time to determine what processes are running. Based on the system information and the process information, the guardian can identify any need to initiate, restart, or stop one or more processes. Further, the guardian process can pass process identification and other process information (i.e., parameters) to an initiated process to assist the process in determining how to proceed.

Parameters specific to each individual server can be defined in a process paramaters table. Further, generic process parameters can be stored in the parameters table. Once a process is initiated, it is responsible for fetching any parameters in the parameters table. Further, each process can determine the frequency at which to refresh the values for its parameters. FIG. 13C provides an example of a process parameters table.

ClassId **1354** has the same meaning as previously described. InstanceId **1356** identifies a process instance as previously described, or identifies that the record contains generic, class parameters. That is, a null or zero value for instanceId **1356** indicates that the corresponding record contains class level parameters. These generic parameters

can be overridden by specific parameters (i.e., parameters specific to a process instance). Parameter **1358** identifies a particular parameter. The valueNum **1360**, valueChar **1362**, and valueDate **1364** fields contain the actual parameter values (i.e., of type number, character, and date, respectively).

Each parameter for a process is paired with an identifying token. Tokens are stored in the process tokens table. Tokens can be described, for example, by the mail administrator. FIG. **14**A provides an example of a process tokens table. ClassId **1404** has the same description as previously described. ParameterId **1406** identifies a particular parameter. ParameterType **1408** identifies parameter types (e.g., number, character and date). Name **1410** can be used to identify the token in a display. The description field **1412** can be used to provide a description or commentary for a token.

Each process has associated record(s) in the process time table. The process time table is used to manage the wake and sleep times for a process instance. Process time table records can be used by an instantiated process to determine its actual requested Active and Passive (i.e., sleeping) times. FIG. **14**B provides an illustration of a process time table.

The classId field **1434** and instanceId field **1436** are the same as the similarly-named fields in the previously described tables. StartTime **1438** contains the value that identifies when a process begins its current state. The duration field **1440** indicates the length of time that a process is to remain in a state (e.g., active or dormant). The process will compare the startTime and duration values and the current time to determine whether or not it is to change states.

The flags field **1442** is used to specify the desired state during this designated time. For example, a flags value may indicate active to specify that the associated process is to be active at this time, or it may be used to indicate passive to specify that the process is meant to be dormant during this time. Processes can be tuned with this parameter having a different value during different times of the day. The sleepTime field **1444** indicates the delay (e.g., in minutes) between cycles. The state field **1446** indicates the state of the process (e.g., active or passive). The runIndex **1448** indicates a run state that is examined for changes.

—Mail Objects—

As previously indicated, mail objects (e.g., messages) can be retained in tables in an RDBMS. Tables can indicate one or more queues to which a mail object belongs. A message can be contained in more than one queue. For example, a message sent to both local and remote users can be contained in multiple queues (e.g., a local delivery queue and remote delivery queue). Further, information associated with mail objects can be stored in tables such as an instance table. An instance table entry contains information associated with a message instance. FIG. **15** provides an example of an instance table.

Each object is identified by an identifier that is unique at each node. The msgId **1504** provides this unique identification. Using a unique message identifier, for example, provides the ability to relate additional mail object information in other tables with a given mail object. FolderId **1506** provides ownership and location information. For example, a user's inbox value is stored in the folderId field value for new, unread or read messages. Or a gateway outbox value is stored in the folderId field for a message awaiting submission to a gateway.

A priority field **1508** identifies a mail object's priority. As previously indicated, the priority can be used to determine the order in which mail objects are processed. The flags field **1510** provides additional information associated with a mail object. For example, whether or not the owner of a message is a blind carbon copy recipient. The retentionDate and receivedDate fields (i.e., **1512** and **1514**, respectively) provide time stamp information that can be used, for example, in garbage collection or as the entry time of a message in a queue. Status **1516** indicates the state of a mail object (e.g., new or unread).

The queue field **1518** defines the queue in which the associated mail object instance resides. This field can be examined by a process to determine the mail objects to be processed in a particular queue. For example, a postman process that is configured to perform a notification task may examine the instance table to identify objects in the notification queue that are to be processed.

Guardian

A guardian process determines the number and type of processes to initiate based on configuration information supplied by the mail system administrator. In the preferred embodiment, this information is stored in relations in an RDBMS as previously described. However, any method of retaining configuration information can be used with the present invention.

Further, the guardian process retains a snapshot of current processes, and can obtain a new snapshot. Based on a comparison of the two snapshots and the configuration information, the guardian can determine whether or not to initiate, restart, or stop one or more processes. Further, the guardian process can pass process identification and other process information to a process. A guardian process can act as the parent process for other processes. It spawns or terminates a process after it verifies the proper number of each process type.

FIG. **2** illustrates a process flow for a guardian process. At decision block **202** (i.e., "any signal from a child process?"), if there is no signal from a child process, processing continues at block **204** to block any restart signals and to get any previously generated restart or terminate signals that have not been processed. Processing continues at decision block **206**. If a signal is received from a child process, processing continues at decision block **206**.

At decision block **206** (i.e., "terminate signal?"), if the signal is a terminate signal, processing continues at block **208** to mark all processes spawned by the guardian as obsolete (i.e., terminable). Processing continues at processing block **210**. If, at decision block **206**, the signal is not a terminate signal, processing continues at block **210** to invoke GuardianInit to, for example, generate a new process snapshot. At block **212**, terminateProcess is invoked to kill the appropriate processes. At block **214**, startProcess is invoked to start the appropriate processes.

At decision block **216** (i.e., "any child processes still running?"), if there are no spawned processes running, processing ends at block **218**. If spawned processes are running, processing continues at block **220** to unblock the restart signal. At block **222**, guardian waits for a signal. Signals can be generated by a child or as a result of system administrator input. When guardian receives a signal, processing continues at block **224**. At block **224**, findChild is invoked to identify the processed associated with the signal generator. At decision block **226** (i.e., "child found?"), if the signal generator is unknown, processing continues at block **216**.

If, at decision block **226**, the signal generator is identified, processing continues at block **228**. At block **228**, the

respawn variable is set to include the run and restart alternatives. At decision block **230** (i.e., "child terminated and configured to run if it terminates?"), if a terminated process is configured to be restarted upon termination, processing continues at block **232** to reset respawn to indicate "stateRun" and processing continues at decision block **234**. If not, processing continues at decision block **234**.

At decision block **234** (i.e., "process state for child process & respawn=respawn"), if the state of a child process (i.e., signal generator) is to be respawned based on the value of the respawn variable, processing continues at block **236** to invoke spawnChild. Processing continues at decision block **216**. If it is determined that the child process is not intended to be respawned, processing continues at decision block **216**.

—GuardianInit—

GuardianInit is invoked in the guardian process flow to, for example, fetch a new process snapshot from the RDBMS. FIG. **3** illustrates a guardianInit process flow. At block **302**, an RDBMS connection is established. At block **304**, the number of processes in the process table is determined. This count can be used, for example, for memory allocation purposes. As illustrated in block **306**, the count is used to allocate any additional memory for the process information data structures stored in memory and accessed by the guardian process.

At block **308**, a new process snapshot is fetched from the process table. At block **310**, the restart bit in the processState field of the process table is turned off. At block **312**, a node state variable is set to "shut down." At decision block **314** (i.e., "at least one process in table with state='run'?"), if the snapshot contains at least one process that is to be run, processing continues at block **316** to set the node state variable to "operational," and processing continues at block **318**. If not, processing continues at block **318**. At block **318**, the state of the node is set to the node state variable. Processing returns at block **320**.

—TerminateProcess—

Process termination can, for example, occur when it is determined that a surplus of processing capability exists for a given queue. For example, a mail system administrator monitoring system activity may determine that a queue that is being managed, or handled, by two Postman processes, can be managed by one Postman process. The system administrator can generate a signal for the guardian to terminate one of the Postmen. Further, processes may be terminated when shutting down a mail system.

FIG. **4** illustrates a terminateProcess process flow. At decision block **402** (i.e., "all processes in process data structure processed?"), if all of the processes in the guardian's process snapshot have been processed, processing returns at **404**. If not, processing continues at processing block **406**. At block **406**, the next spawned process' classId and instanceId are identified. At block **408**, findProcess is invoked to locate a snapshot entry corresponding to the spawned process' classId and instanceId.

At decision block **410** (i.e., "process not found or process state=restart or process state<>run?"), if the entry in the process snapshot is to run or be restarted upon termination, or was not found, processing continues at decision block **402** to process any remaining children. If not, processing continues at block **412** to terminate the execution of the process. Processing continues at decision block **402** to process any remaining processes.

—StartServer—

Processes can be initiated by a guardian process, for example, upon a system startup, or when additional processing capability is needed to handle an increase in mail activity. FIG. **5** illustrates a startServer process flow. At decision block **522** (i.e., "all processes in table memory processed?"), if all processes have been processed, processing returns at block **524**. If not, processing continues at block **526** to get the next process in the process snapshot.

At decision block **528** (i.e., "process state='run'?"), if the state of the process is not set to run, processing continues at decision block **522** to process any remainder of the processes. If the process state is equal to run, processing continues at decision block **530**. At decision block **530** (i.e., "process already running?"), if the process is already running, processing continues at decision block **522** to process the remaining processes. If not, processing continues at decision block **532**.

One technique for determining whether or not a process is already running, involves maintaining a list of executing processes and their associated class and instance identifications. Thus, the list of executing processes can be examined to determine whether or not a process is already running. Any other method can be used without departing from the scope of the invention.

At processing block **532**, spawnProcess is invoked to initiate the process. Processing continues at decision block **522** to process any remaining snapshot entries.

—findChild—

The findChild process can associate an executing process with a process snapshot entry based on like process information (e.g., classId and instanceId). It can be invoked, for example, to determine which child process terminated. FIG. **6** illustrates a findChild process flow. At decision block **602** (i.e., "all entries in spawned process list checked?"), if all entries in a list of executing processes has been processed, processing returns at **604** with a return code to indicate that no child process was found (e.g., a negative one).

If all entries have not been processed, processing continues at block **606** to obtain the next entry in the list. At decision block **608** (i.e., "child's pid=desired pid?"), if the entry's process identification (e.g., process identification generated by the operating system when the process was initiated) is not the same as the desired pid (e.g., the pid accompanying the process' termination signal), processing continues at decision block **602** to examine the remaining entries in the spawned process list.

If it is the same, processing continues at block **610** to delete the entry from the spawned process list. At block **612**, findProcess is invoked to identify the terminated process' entry in the process snapshot. The location of the process in the process snapshot is returned at block **614**.

—findProcess—

An entry in the process snapshot can be obtained using the findProcess flow illustrated in FIG. **7**. At decision block **702** (i.e., "all snapshot entries checked?"), if all snapshot entries have been checked, processing returns at **704** with a return code to indicate that no snapshot entry was found (e.g., a negative one). If not, processing continues at block **706** to get the next snapshot entry.

At decision block **708** (i.e., "current snapshot entry's classId and instanceId=desired class and instance?"), if the current entry has the same class and instance identification

as the desired class and instance information, processing returns at block **710** with a snapshot entry identification. If not, processing continues at decision block **702** to process any remaining snapshot entries.

—spawnProcess—

FIG. **8** illustrates a process flow, spawnProcess, for spawning a child process. At block **802**, the process state is set to "restart." At block **804**, a parameter list (e.g., classId and instanceId) is generated to pass to the spawned process. The process is spawned (e.g., using a fork operation in Unix) and the parameters are passed to the new process at block **806**. At block **808**, information associated with the spawned process is stored in the spawned process list (e.g., class, instance, spawned process' system identification). At block **810**, the spawnedProcess counter is incremented by one. Processing returns at block **812**.

Spawned Process

Different types of processes can be used in the present invention to perform mail system tasks. A process' configuration is determined from the information in the process tables (e.g., process, class, process parameters, process time, and process tokens tables). A configuration can, for example define the tasks to be performed by a spawned process. Examples of types of process' and associated tasks have been described previously. Additional or different process types and tasks can be used without departing from the scope of this invention.

As previously described, a postman process can deliver local mail objects, remote mail objects, handle triggered mail objects (e.g., return receipts and auto-forward), and send local users notification of new messages. FIG. **9A** illustrates a postman process flow. At block **902**, passCount is set to zero and exitCode is initialized to "Restart." At block **906**, any parameters sent by the invoking process are obtained. At block **910**, checkState is invoked to determine the state of a process and system.

At decision block **912** (i.e., "system shutdown or restarted?"), if the system was shutdown or restarted, processing continues at block **914** to set the exitCode to "noRestart," and processing continues at block **926**. If not, processing continues at decision block **916**. At decision block **916** (i.e., "runFlag='Active'?"), if the flag indicates that the process is not meant to be active during this period, processing continues at decision block **940**.

If the process is meant to be active, processing continues at decision block **918**. At decision block **918** (i.e., "server's actions contained in command-line argument?"), if configuration information was passed to the process, processing continues at block **922**. If not, processing continues at block **920** to get the postman flags for this process from the process parameters table. At block **922**, the process table's wakeTime timestamp associated with this process is updated. At block **924** performServerAction is invoked to perform the tasks configured for this process. Processing continues at decision block **940**.

—Dormancy—

A process can be configured to sleep after completing a processing pass or cycle wherein the process attempts to perform tasks for which it is configured to perform. As illustrated in the Postman process flow, the dormancy stage of processing can be initiated when the postman completes one cycle of processing, or when the process' flags indicates that the process is meant to be dormant.

FIG. **9C** illustrates the dormancy preparation for the postman process. At decision block **940** (i.e., "run multiple passes?"), if a process is not meant to run more than one pass, or cycle, processing continues at block **942** to set exitCode to "noRestart" and processing continues at block **926**. If the process is configured to run multiple passes, processing continues at block **943**. At block **943**, a passCount is incremented by one. At decision block **944** (i.e., "passCount=number of passes desired?"), if the configured number of passes have been performed, processing continues at block **946** to set exitCode to "noRestart" and processing continues at block **926**.

If the number of multiple cycles has not been achieved, processing continues at block **948** to invoke checkState. At decision block **950** (i.e., "system shutdown or restarted?"), if the system was shutdown or restarted, processing continues at block **952** to set the exitCode to "noRestart," and processing continues at block **926**. If not, processing continues at decision block **954**. At decision block **954** (i.e., "delay time contained in command-line arguments?"), if the delay time was passed to the process, processing continues at block **956** to assign the passed value to the delayTime variable, and processing continues at decision block **968**.

If not, processing continues at decision block **958**. At decision block **958** (i.e., "runFlag='Active'?"), if the flag indicates that the process is not meant to be active during this period, delayTime is set to the time remaining until it is to be active, and processing continues at decision block **968**. If the flag indicates that the process is meant to be active, processing continues at decision block **962**.

At decision block **962** (i.e., "specific delay time set for this server?"), if there is a delayTime value for this process, processing continues at block **966** to set the delayTime variable to this time. Processing continues at decision block **968**. If not, a default delayTime is used at block **964**, and processing continues at decision block **968**.

At decision block **968** (i.e., "reLogin indicator contained in command-line arguments?"), if the indicator for logging back into the RDBMS was passed to the process, processing continues at block **970** to use the passed value to set the reLogin variable. Processing continues at block **978**. If not, processing continues at decision block **972**.

At decision block **972** (i.e., "specific reLogin indicator set for this type of server?"), if a specific reLogin value is set, this value is used to set the reLogin variable at block **976**, and processing continues at block **978**. If not, processing continues at block **974** to use a default for this class of process, and processing continues at block **978**.

At block **978**, the time at which the process becomes dormant (i.e., the sleepTime timestamp) is stored in the associated entry in the process table. At decision block **980** (i.e., "reLogin="Yes"?"), if reLogin is positive, registration of the process is deleted, and processing continues at block **984**. If it is negative, processing continues at block **984**. At block **984**, the process becomes dormant for the number of minutes determined by the value of delayTime. After awaking, the process continues at block **910** to determine the state of the system.

—Process Completion—

As previously indicated, a process can complete after one or more cycles, or upon some other indication. FIG. **9B** illustrates a Postman process flow anticipating process completion. At block **926**, work areas are freed. At decision block **928** (i.e., "connected to database?"), if the process is not connected to the database, processing continues at block

934. If it is, processing continues at block 930 to rollback any uncommitted transactions. Processing continues at block 934.

At block 934, any space acquired during processing (e.g., contexts) is freed. Processing returns at block 936 to the invoking process with the value of exitCode. The exitCode value can be used by the invoking process (e.g., guardian) to determine whether or not this process is to be restarted.

—checkState—

The checkState process checks the state of the system and a particular process. FIG. 10 provides an example of a checkState process flow. At block 1002, either specific or generic (where there are no specific values) startTime, duration, flags, sleepTime, processState, and runIndex values are obtained from the procTime table. At block 1004, runFlag variable is updated from the flags value. The remainingTime is determined from the startTime and duration values at block 1006. At decision block 1012 (i.e., "system shutdown or restarted, or change in runIndex?"), if the system has shutdown or been restarted or there has been a change in runIndex, processing returns a warning at block 1014. If not, processing returns a successful value at block 1016.

—performServerAction—

FIG. 11 illustrates a performServerAction process flow for a Postman process. At decision block 1102 (i.e., "remote flag set?"), if the remote flag is set, the Postman performs remote instance deletion at processing block 1104 and processing continues at decision block 1106. If not, processing continues at decision block 1106. At decision block 1106 (i.e., "gateway flag set?"), if the gateway flag is set, the Postman performs gateway processing at processing block 1108 and processing continues at decision block 1110. If not, processing continues at decision block 1110.

At decision block 1110 (i.e., "local flag set?"), if the local flag is set, the Postman performs message forwarding to remote sites at processing block 1112 and processing continues at decision block 1114. If not, processing continues at decision block 1114. At decision block 1114 (i.e., "local flag set?"), if the local flag is set, the Postman performs local message delivery at processing block 1116 and processing continues at decision block 1118. If not, processing continues at decision block 1118.

At decision block 1118 (i.e., "notification flag set?"), if the notification flag is set, the Postman performs notification at processing block 1120 and processing continues at decision block 1122. If not, processing continues at decision block 1122. At decision block 1122 (i.e., "local flag set?"), if the local flag is set, the Postman performs local message forwarding at processing block 1124 and processing continues at decision block 1124. If not, processing continues at decision block 1126.

At decision block 1126 (i.e., "remote flag set?"), if the remote flag is set, the Postman performs remote message delivery at processing block 1128 and processing continues at decision block 1130. If not, processing continues at decision block 1130. At decision block 1130 (i.e., "gateway flag set?"), if the gateway flag is set, the Postman performs non-local delivery of remote messages to gateway nodes at processing block 1132 and processing returns at decision block 1134. If not, processing returns at decision block 1134.

Mail Object Locking

To perform configured tasks, one or more processes must access records in a queue. In the preferred embodiment, queue entries can be stored in a database with each queue entry being a row in a database relation, or table (e.g., the instance table). As each entry is selected for processing, the row in the table that corresponds to the queue entry can be locked. Each process can examine a snapshot of the queue and attempt to access the next queue entry. If the entry is not locked, the entry can be selected for processing. If the entry is locked, the entry cannot be selected by a subsequent, inquiring process.

FIG. 11 includes an example of a postman's local message delivery configurable task. FIG. 12 illustrates a local message delivery process flow including locking. At block 1202, message and folder identification is obtained from the instance table where the queue is a specified queue value and the retentionDate is less than or equal to the system date. The selected messages are ordered by priority and retentionDate.

The number of messages obtained is set in msgCount at processing block 1204. At decision block 1206 (i.e., "all selected messages processed?"), if all of the selected messages are processed, processing returns the number of successful deliveries at block 1208. If not, processing continues at block 1212 to select the next mail object from those selected. A selection for update causes the message to be locked from other access. Further, such a selection locks out subsequent attempts to access the record. Thus, at decision block 1214 (i.e., "locking error?"), if a locking error occurs, the process logs the error and processing continues at decision block 1206 to select another from any remaining messages.

If a locking error does not occur, processing continues at decision block 1218 to handle other exceptions. If it is determined that the message has already been delivered, processing continues at block 1220 to generate a log entry. Processing continues at decision block 1206 to process any remaining messages. If not, processing continues at decision block 1222 (i.e., "other error?"), if some other, unknown error occurs, processing returns at block 1238.

If no locking or other error occurs, processing continues at block 1226, to deliver the message. At block 1228, a log entry is generated to log the status of the delivery. At decision block 1230 (i.e., "delivery error?"), if a delivery error did not occur, processing continues at block 1234 to increment the successful delivery counter, and processing continues at decision block 1206. If a delivery error occurred, processing continues at block 1232 to unblock the message, and processing continues at decision block 1236.

At decision block 1236 (i.e., "fatal?"), if the error is a fatal error, processing returns at block 1238. If not, processing continues at decision block 1206 to process any remaining messages.

Thus, a method and apparatus for processing electronic mail in parallel has been provided.

What is claimed is:

1. A method for processing electronic mail objects in parallel in a computer system, the method comprising the computer-implemented steps of:

receiving electronic mail objects in a data structure;

a first process spawning child processes;

said child processes processing said electronic mail objects in parallel;

a second process detecting a change in the number of said electronic mail objects in said data structure;

a third process spawning at least one additional child process in response to an increase in the number of said electronic mail objects in said data structure;

a fourth process terminating at least one of said child processes in response to a decrease in the number of said electronic mail objects in said data structure; and

at least one of said child processes processing additional electronic mail objects until terminated by said fourth process.

2. The method of claim **1** wherein said step of spawning child processes, said step of detecting a change in the number of said electronic mail objects, said step of spawning at least one additional child process and said step of terminating at least one of said child processes are performed by a single process.

3. The method of claim **1** wherein said step of receiving electronic mail objects in a data structure comprises the step of queuing said electronic mail objects in a queue.

4. The method of claim **1** wherein said step of detecting a change in the number of said electronic mail objects comprises the step of detecting a change in the size of said data structure.

5. The method of claim **1** wherein said step of processing comprises the steps of one of said child processes:

selecting one of said electronic mail objects from said data structure;

locking said electronic mail object to prevent selection thereof by another of said child processes;

processing said electronic mail object; and

removing said electronic mail object from said data structure.

6. The method of claim **5** wherein said step of selecting comprises the step of selecting an electronic mail object having a higher priority than others of said electronic mail objects.

7. The method of claim **5** wherein said step of selecting comprises the step of selecting an electronic mail object placed in said data structure before others of said electronic mail objects.

8. The method of claim **5** wherein said step of placing electronic mail objects in a data structure comprises the step of storing each of said electronic mail objects in a record of a data base and said step of locking said electronic mail object comprises the step of locking the record in which said electronic mail object is stored.

9. In an electronic mail system, a method of dynamically scaling processing capability comprising the steps of:

said electronic mail system receiving electronic mail objects;

a first process spawning child processes;

said child processes processing said electronic mail objects in parallel;

a second process configured to detect an increase in processing activity of said child processes and to spawn at least one additional child process in response to detecting the increase; and

a third process configured to detect a decrease in processing activity of said child processes and to terminate at least one of said child processes in response to detecting the decrease.

10. The method of claim **9** wherein said step of spawning child processes, said step of detecting an increase in processing activity and said step of detecting a decrease in processing activity are performed by a single process.

11. A data processing system having at least one data processing node, said data processing system comprising:

a data structure on said data processing node receiving electronic mail objects;

child processes processing said electronic mail objects in parallel;

a first process detecting a change in the number of said electronic mail objects in said data structure;

a second process spawning an additional child process in response to an increase in the number of said electronic mail objects;

a third process terminating a child process in response to a decrease in the number of said electronic mail objects; and

at least one of said child processes processing additional electronic mail objects until terminated by said third process.

12. The data processing system of claim **7** wherein said first process is configured to detect said change in the number of said electronic mail objects by detecting a change in the size of said data structure.

13. The data processing system of claim **7** wherein a single process comprises said first process, said second process and said third process.

14. An article of manufacture including one or more computer readable media having program code stored thereon, the program code including instructions which, when executed by a processor, cause the processor to:

spawn child processes to process in parallel electronic mail objects received in a data structure;

detect a change in the number of the electronic mail objects in the data structure;

spawn at least one additional child process in response to an increase in the number of the electronic mail objects in the data structure; and

terminate at least one of the child processes in response to a decrease in the number of the electronic mail objects in the data structure, at least one of the child processes processing additional electronic mail objects until terminated.

15. The article of claim **14** wherein detecting the change in the number of the electronic mail objects includes detecting a change in the size of the data structure.

16. The article of claim **14** wherein spawning child processes to process the electronic mail objects includes spawning at least one child process to:

select one of the electronic mail objects from the data structure;

lock the selected electronic mail object to prevent selection thereof by another of the child processes;

process the selected electronic mail object; and

remove the selected electronic mail object from the data structure.

17. An article of manufacture including one or more computer readable media having program code stored thereon, the program code including instructions which, when executed by a processor, cause the processor to:

spawn child processes to process electronic mail objects in parallel;

detect an increase in processing activity of the child processes;

spawn at least one additional child process in response to detecting the increase;

detect a decrease in processing activity of the child processes; and

terminate at least one of the child processes in response to detecting the decrease.

18. A computer system including a processor and a memory coupled to the processor, the memory having stored

therein a data structure to receive electronic mail objects, and program code which, when executed by the processor, causes the processor to:

spawn child processes to process the electronic mail objects in the data structure in parallel;

detect a change in the number of the electronic mail objects in the data structure;

spawn at least one additional child process in response to an increase in the number of the electronic mail objects in the data structure; and

terminate at least one of the child processes in response to a decrease in the number of the electronic mail objects in the data structure, at least one of the child processes processing additional electronic mail objects until terminated.

19. The computer system of claim **18** wherein detecting the change in the number of the electronic mail objects includes detecting a change in the size of the data structure.

20. The computer system of claim **18** wherein spawning child processes to process the electronic mail objects includes spawning at least one child process to:

select one of the electronic mail objects from the data structure;

lock the selected electronic mail object to prevent selection thereof by another of the child processes;

process the selected electronic mail object; and

remove the selected electronic mail object from the data structure.

* * * * *